Redesigning Lisp-Stat Around an Abstract Statistical Machine

Luke Tierney
School of Statistics
University of Minnesota
Minneapolis, MN 55455
luke@umnstat.stat.umn.edu

Background

Lisp-Stat is a statistical computing environment based on the Lisp language (a subset of Common Lisp).

Lisp-Stat includes

- vectorized arithmetic
- basic statistics and linear algebra functions
- a window system interface
- access to user interface tools
- dynamic graphics tools
- an object-oriented programming system

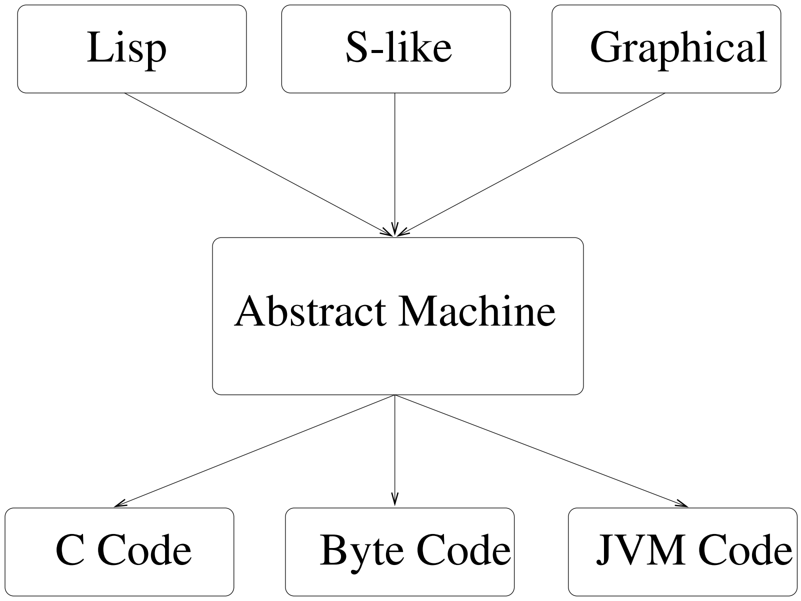
Current System Design

- Originally interpreted in C code
- Added byte code compiler
  Converts list structure into a stream of bytes;
  also simplifies code:
  - eliminates variable lookup
  - eliminates unused features
  - inlining
  - tail recursive

New Directions

- new abstract machine and runtime
- using/working with Java
- multithreading
- distributed computing
- revised graphics model
- alternate languages

Abstract Machine Design



- multiple implementations
  - C byte code interpreter
  - native code from C
  - Java byte code
- language-independent core
  - Lisp syntax
  - S-like syntax
  - graphical syntax

Abstract Machine Components

<table>
<tr><td colspan="3">Machine Opcodes</td><td>Library Interface</td><td>...</td></tr>
<tr><td colspan="5">...</td></tr>
<tr><td>Heap</td><td>Threads</td><td>Procedure Calling</td><td>Exceptions</td><td>...</td></tr>
</table>

Machine Opcodes

Examples

Lisp Code	
`(lambda (x y)` `  (+ (f x) y))`	`(lambda (x y)` `  (f (+ x y)))`

Intermediate Code	

`(lambda nil (x y)`
`  (call f (x) (tmp)`
`    (primop + (tmp y))))`
	`(lambda nil (x y)`
	`  (primop + (x y) (tmp))`
	`    (call F (tmp)))`

Pseudo-Assembly Code	

`(%initialize-0 2)`
`(%call f x tmp)`
`(%tail-primop + tmp y)`
	`(%initialize-0 2)`
	`(%primop + x y tmp)`
	`(%tail-call f tmp)`

Opcode Classes

- High level opcodes—`%call`, `%initialize-0`
- Utility opcodes—`%goto`
- Basic data operations—`+`, `aref`
- Special data operations—`car`, `cdr`
- Vector/array operations
  - `VCODE` opcodes from NESL
  - direct support for BLAS.

Object Support

To support opjects “all the way down” almost all opcodes handle default cases and dispatch for others:

```
Node primopAdd(State s, int depth, Node x, Node y) {
  if (isFixnum(x) && isFixnum(y))
    return FixnumAdd(x, y);
  else if (isReal(x) && isReal(y))
    return newFlonum(doubleValue(x) + doubleValue(y));
  else return TAILCALL(standardAdd, s, d, x, y);
}
```

Procedure Representation

Issues

- variable number of arguments
- keyword arguments (of some form)
- multiple return values
- tail recursion

Supporting Tail Recursion

Functional languages often use recursion for iteration:

```
let sum(s, x, i, n)
      if i < n then sum(s+x[i],i+1,n)
      else s
in
  sum(0, x, 0, length(x))
```

This does not work in C/Java:

> Both (usually) allocate activation records for all calls (including tail calls) on the stack.

To support tail recursion in Java, define a `State` class that represents registers,

```
class State {
  Procedure p;
  int i0, i1, i2, i3;
  Object a0, a1, a2, a3;
}
```

and an abstract `Procedure` class that contains the tail call mechanism.

```
abstract class Procedure {
  static final int MAXDEPTH = 10;
  abstract void call(State s, int depth);
  static void tailcall(Procedure p, State s, int depth) {
    if (depth == 0)
      do {
        s.p = null;
        p.call(s,1);
      } while ((p = s.p) != null);
    else if (depth > MAXDEPTH)
      s.p = p;
    else
      p.call(s, depth + 1);
  }
}
```

The `sum` example then looks like

```
class Sum extends Procedure {
  void call (State s, int depth) {
    int sum = s.i0, i = s.i1, n = s.i2;
    int[] x = (int[]) s.a0;
    if (i < n) {
      s.i0 = sum + x[i];
      s.i1 = i + 1;
      tailcall(this, s, depth);
    }
    else s.i0 = sum;
  }
  public static void main(String[] argv) {
    int n = Integer.parseInt(argv[0]);
    int[] x = new int[n];
    for (int i = 0; i < n; i++)
      x[i] = i + 1;
    State s = new State();
    Procedure p = new Sum();
    s.i0 = 0; s.i1 = 0; s.i2 = n;
    s.a0 = x;
    p.call(s, 0);
    System.out.println("Value = " + s.i0);
  }
}
```

To reduce overhead of calling `tailcall`, can inline the `depth` tests:

```
class Sum1 extends Procedure {
  void call (State s, int depth) {
    int sum = s.i0, i = s.i1, n = s.i2;
    int[] x = (int[]) s.a0;
    if (i < n) {
      s.i0 = sum + x[i];
      s.i1 = i + 1;
      if (depth == 0 || depth > MAXDEPTH)
        tailcall(this, s, depth);
      else
        this.call(s, depth + 1);
    }
    else s.i0 = sum;
  }
  public static void main(String[] argv) {
    int n = Integer.parseInt(argv[0]);
    int[] x = new int[n];
    for (int i = 0; i < n; i++)
      x[i] = i + 1;
    State s = new State();
    Procedure p = new Sum();
    s.i0 = 0; s.i1 = 0; s.i2 = n;
    s.a0 = x;
    p.call(s, 0);
    System.out.println("Value = " + s.i0);
  }
}
```

Heap for C

- supports finalization, weak references
- non-copying (simplifies native code interface)
- allocation by pages
- separate heaps per thread
- shared heap for shared data
- thread heaps collected by implicit collector
- shared heap collected with all threads suspended
- better: incremental concurrent collector (Doligez and Leroy, Doligez and Gonthier)

Other Issues

Exceptions

- Exceptions can be used for non-local exits, but may be slow (need to turn off stack trace).
- Java exceptions don't allow handling in context (CL, Dylan, Tom , signals)

Multithreading

- user-level multithreading
- multiple OS threads, processors
- CSP-style channels as main synchronization approach
- low level synchronization (mutexes/semaphores) if needed

Next Steps

- Exceptions for C
- Threads for C
- Flesh out opcodes and implement on C, Java
- resolve library interface issues
- Port current Lisp-Stat

Why Compilation

- Efficiency
  - user code will run faster
  - less native system code needed
- Helps clean up the language
  - features that are hard to compile are hard to understand
  - thinking about compilation helps to clarify language semantics