Compilers are from Mars, Dynamic Scripting Languages are from Venus

Jose Castanos, David Edelsohn, Kazuaki Ishizaki, Priya Nagpurkar, Takeshi Ogasawara, Akihiko Tozawa, and Peng Wu
Compilation for Dynamic Scripting Languages

- **Dynamic scripting languages** (DSLs) are gaining popularity, and start to be used for production development

<table>
<thead>
<tr>
<th>Commercial deployment</th>
<th>Cloud</th>
</tr>
</thead>
<tbody>
<tr>
<td>Facebook (PHP)</td>
<td>Google AppEngine (Python)</td>
</tr>
<tr>
<td>YouTube (Python)</td>
<td></td>
</tr>
<tr>
<td>Invite Media (Python)</td>
<td></td>
</tr>
<tr>
<td>Twitter (Ruby on Rails + Scala)</td>
<td></td>
</tr>
<tr>
<td>ManyEyes (Ruby on Rails)</td>
<td></td>
</tr>
</tbody>
</table>

- Opportunities
  - DSL adoption happens in areas with great growths that have not traditionally been our market

---

**TIOBE Language Index**

<table>
<thead>
<tr>
<th>Rank</th>
<th>Name</th>
<th>Share</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>C</td>
<td>16.986%</td>
</tr>
<tr>
<td>2</td>
<td>Java</td>
<td>16.668%</td>
</tr>
<tr>
<td>3</td>
<td>PHP</td>
<td>10.298%</td>
</tr>
<tr>
<td>4</td>
<td>C++</td>
<td>8.554%</td>
</tr>
<tr>
<td>5</td>
<td>Basic</td>
<td>6.757%</td>
</tr>
<tr>
<td>6</td>
<td>C#</td>
<td>5.444%</td>
</tr>
<tr>
<td>7</td>
<td>Python</td>
<td>5.179%</td>
</tr>
<tr>
<td>8</td>
<td>Perl</td>
<td>3.474%</td>
</tr>
<tr>
<td>9</td>
<td>Ruby</td>
<td>2.370%</td>
</tr>
<tr>
<td>10</td>
<td>JavaScript</td>
<td>2.149%</td>
</tr>
</tbody>
</table>

“Python helped us gain a huge lead in features and a majority of early market share over our competition using C and Java.”

- Scott Becker, CTO of Invite Media
  Built on Django, Zenoss, Zope

---

Compilers are from Mars, and Dynamic Scripting Languages are from Venus © 2010 IBM Corporation
Motivation

- Dynamic scripting languages (DSL)
  - Python, Ruby, PHP, Javascript, Lua, and many others

- Optimization of DSL programs is an active area of research
  - renewed browser wars
    - TraceMonkey (Mozilla), SPUR (MS), V8 (Google)
  - cloud deployment
    - AppEngine from Google

- Significantly slower compared to equivalent in Java and C
  - mostly interpreted, not highly optimized, richer semantics for basic operations

- The research landscape for DSL compilation is vast
  - no low-hanging fruits for compilation
  - a lot of variability in results
  - no agreed principles in the community
Language Comparison (Shootout)

Languages: Java (JIT, steady-version); Python, Ruby, Javascript, Lua (Interpreter)
Standard DSL implementation (interpreted) can be 10~100 slower than Java (JIT)
Python Language and Implementation

- Python is an object-oriented, dynamically typed language
  - also support exception, garbage collection, function continuation

```python
def foo(list):
    return len(list)+1
```

Python bytecode:

- LOAD_GLOBAL (name resolution)
  - dictionary lookup

- CALL_FUNCTION (invocation)
  - frame object, argument list processing, dispatch according to types of calls

- BINARY_ADD (type generic operation)
  - dispatch according to types, object creation

All three involve layers of runtime calls (via function pointers), reference counting, and exception checking
Compilers are from Mars, and Dynamic Scripting Languages are from Venus
Python Implementations (Unladen-Swallow benchmarks)

![Bar chart showing speedup and slowdown of different Python implementations relative to CPython. The chart compares Unladen-Swallow, PyPy, IronPython, and Jython/HotSpot 7.]
Python Implementations (shootout benchmarks)

![Graph showing speedup and slowdown of different Python implementations relative to CPython.](image)
A System View of Optimizing DSL Compilers

Optimizing DSL compiler

- DSL semantics (dynamic typing, rich operators and built-in types, etc)
- Traditional IR semantics (static typing, etc)
- Backend IR semantics (register, instr, etc)
- Machine semantics (ISA)

DSL program

- Optimizer
- Translation (IR-gen)
- Machine binary

DSL interpreter + runtime
An Optimization Example (LOAD_GLOBAL)

100 SETUP_LOOP;
    LOAD_GLOBAL 1 ('foo');
    CALL_FUNCTION;
    ...
    JUMP_ABSOLUTE 100;

**DSL data-flow optimizer** (e.g., bytecode optimizer): hoist LOAD_GLOBAL out of the loop if one can prove it invariant.

**Translation time optimization** (e.g., unladen-swallow): in-line caching with guards, 3%~9% improvements on rietfield, django, 2to3 from unladen-Swallow benchmarks.

**Optimization inside runtime** (e.g., jython): improve dictionary (hashtable) lookup by inlining, code straightening, etc.

23 python BC
252 java BC
148 nodes and 5 BBs in initial IR
6882 nodes and 769 BBs after inlining 178 sites
Optimizing Compiler Approaches

Unladen Swallow

- DSL semantics
- Optimizer
- Translation

LLVM (C)
- Binary
- Backend
- Legacy target language: C/C++

Jython

- DSL semantics
- Optimizer
- Translation

Java JIT
- Backend
- Legacy target language: Java

PyPy

- DSL semantics
- Optimizer
- Translation

RPython JIT
- Backend
- RPython is closest to Python

IronPython

- DSL semantics
- Optimizer
- Translation

CLR JIT (CIL)
- Backend
- Legacy target language: C#
First Level of Lowering of DSL Semantics

<table>
<thead>
<tr>
<th>Translation</th>
<th>Unladen (naïve)</th>
<th>Jython</th>
<th>Unladen w/ feedback</th>
<th>PyPy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ref counting; PyBinaryAdd(); error checking;</td>
<td>invokevirtual PyObject._add()</td>
<td>ref counting; inlined PyIntAdd; error checking;</td>
<td>INT_ADD (unboxed)</td>
</tr>
<tr>
<td>LLVM</td>
<td></td>
<td>JVM</td>
<td>LLVM</td>
<td>RPython</td>
</tr>
<tr>
<td>Direct opt effect</td>
<td></td>
<td></td>
<td>Semantic loss</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Compact IR</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>LLVM IR</td>
<td>JVM IR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>JVM IR</td>
<td>LLVM IR</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>RPython IR</td>
<td>RPython IR</td>
</tr>
</tbody>
</table>

RPython: well-typed, unboxed primitive types, class definition unchange after start-up time
Dynamic Scripting Language JIT Landscape

- **Client**
  - JavaScript
    - V8
    - Squirrel Fish
  - Java
    - DaVinci Machine
  - Ruby
    - Ruby
  - CLR/DLR
    - CLR/DLR

- **Client/Server**
  - Python
    - CLR/DLR
  - JavaScript
    - Unladen-Swallow
  - Ruby
    - DaVinci Machine
  - CLR/DLR
    - CLR/DLR

- **Server**
  - PHP
  - Zend

**Significant difference in JIT effectiveness across languages**
- Javascript has the most effective JITs
- Ruby JITs are similar to Python’s

- **JVM based**
  - Jython
  - JRuby
  - Rhino

- **CLR based**
  - IronPython
  - IronRuby
  - IronJscript
  - SPUR

- **Add-on JIT**
  - Unladen
  - Rubinius

- **Add-on trace JIT**
  - PyPy
  - LuaJIT
  - TraceMonkey
  - SPUR
Concluding Remarks (Questions)

Questions for the community

1. What are the right level(s) to optimize dynamic scripting languages?
2. How to introduce DSL semantics into an optimization infrastructure designed for statically typed languages

Our thoughts

- “Naïve” compilation of DSL provides little benefit
- Dynamism and overhead should be reduced at a suitable IR level
  - Semantic lowering can be “lost-in-translation” or real “strength reduction”
  - Exposing the runtime to optimizer can be double-edged sword: optimizing implementation of the semantics instead of the semantics

Our project (FIORANO)

- Explore compilation of Python and other DSL languages
- Build on top of legacy infrastructure (Testarossa JIT) and public domain VM
- Early experiences match with those of unladen-swallow’s
BACK UP
PyPy (Interpreters + JIT)

- A Python implementation written in RPython
  - interface with CPython modules may take a big performance hit

- RPython is a restricted version of Python, e.g., (after start-up time)
  - *Well-typed* according to type inference rules of RPython
  - Class definitions do not change, support single inheritance
  - Numerical and string types use unboxed representations
  - Tuple, list, dictionary are homogeneous (across elements)

- Tracing JIT through both user program and runtime (RPython)

- Optimizations that work well
  - Removal of frame handling
  - Avoid creating temporary objects
  - Optimize attribute and name lookups
IronPython: DynamicSites

- Optimize method dispatch (including operators)
- Incrementally create a cache of method stubs and guards in response to VM queries

```csharp
public static object Handle(object[],
    FastDynamicSite<object, object, object> site1,
    object obj1, object obj2) {
    if (((obj1 != null) && (obj1.GetType() == typeof(int)))
        && ((obj2 != null) && (obj2.GetType() == typeof(int)))) {
        return Int32Ops.Add(Converter(ConvertToInt32(obj1),
                                Converter.ConvertToInt32(obj3));
    }
    if (((obj1 != null) && (obj1.GetType() == typeof(string)))
        && ((obj2 != null) && (obj2.GetType() == typeof(string)))) {
        return = StringOps.Add(Converter(ConvertToString(obj1),
                                Converter.ConvertToString(obj2));
    }
    return site1.UpdateBindingAndInvoke(obj1, obj3);
}
```

- Propagate types when UpdateBindingAndInvoke recompiles stub
Jython

- Clean implementation of Python on top of JVM
  - Generate JVM bytecodes from Python 2.5 programs
    - interface with Java programs; cannot easily support standard C modules
  - Runtime is rewritten in Java, allow JIT optimize user programs and runtime
  - Python built-in objects are mapped to Java class hierarchy
    - allow (virtual) function specialization based on built-in types

- Large code explosion when applying standard JIT optimizations

- Large memory footprint
  - 300-600MB for small programs (~3MB on CPython)

- New InvokeDynamic bytecode in Java7 specification, but still not implemented in Jython
Unladen-swallow

- **Dealing with Dynamism**
  - Caching LOAD_GLOBAL and import
  - Specialized binary and comparison operators, and builtin functions based on runtime feedback
  - Type inference for native types

- **Implementation Improvements**
  - Fast calls
  - Constantish
  - Expose Cpython stack to JIT (LLVM)
  - Omit untaken branches (during IR generation)
Performance of Javascript implementations

- TraceMonkey
- V8
- Rhino

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>TraceMonkey</th>
<th>V8</th>
<th>Rhino</th>
<th>Speedup (relative to Javascript)</th>
</tr>
</thead>
<tbody>
<tr>
<td>binarytrees</td>
<td>2</td>
<td>56</td>
<td></td>
<td>19</td>
</tr>
<tr>
<td>fasta</td>
<td>2</td>
<td>16</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>mandelbrot</td>
<td>2</td>
<td>8</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>nbody</td>
<td>2</td>
<td>4</td>
<td></td>
<td>4</td>
</tr>
<tr>
<td>spectralnorm</td>
<td>2</td>
<td>4</td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>geomean</td>
<td>2</td>
<td>4</td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>
Performance of LuaJIT

- LuaJit2 (beta 4 -O0)
- LuaJit2 (beta 4 -O1)
- LuaJit2 (beta 4 -O2)
- LuaJit2 (beta 4 -O3)

Speedup (relative to interpreter):

- binarytrees
- fasta
- knucleotide-2
- mandelbrot
- nbody
- pi_digits-4
- regexdna-3
- revcomp
- spectralnorm
- geometric mean
Performance of Ruby Implementations

- Ruby 1.9
- Jruby/OpenJDK
- Jruby/Hotspot 7
- Jruby/TR
- Rubinius

Speedup (Relative to Ruby)