Compilers are from Mars, Dynamic Scripting Languages are from Venus

Peng Wu
IBM Research
Compilation for Dynamic Scripting Languages

- Trend in emerging programming paradigms
  - **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>

<table>
<thead>
<tr>
<th>TIOBE Language Index</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rank</td>
</tr>
<tr>
<td>------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</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
Motivation

- Dynamic scripting languages (DSL)
  - Python, Ruby, PHP, Javascript, Lua, R, 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)

Benchmarks: shootout (http://shootout.alioth.debian.org/) measured on Nehalem Languages: Java (JIT, steady-version); Python, Ruby, Javascript, Lua (Interpreter) Standard DSL implementation (interpreted) can be 10~100 slower than Java (JIT)

Compilers are from Mars, and Dynamic Scripting Languages are from Venus

© 2010 IBM Corporation
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
```

- LOAD_GLOBAL (name resolution)
  - dictionary lookup

```
0 LOAD_GLOBAL 0 (len)
3 LOAD_FAST 0 (list)
6 CALL_FUNCTION 1
9 LOAD_CONST 1 (1)
12 BINARY_ADD
13 RETURN_VALUE
```

- 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
Optimizing Compiler Approaches

Unladen Swallow

- DSL semantics
- Traditional optimizer
- Backend
- Legacy target language: C/C++

Jython

- DSL semantics
- Traditional optimizer
- Backend
- Legacy target language: Java

PyPy

- DSL semantics
- Traditional optimizer
- Backend
- RPython JIT
- RPython is closest to Python

IronPython

- DSL semantics
- Traditional optimizer
- Backend
- CLR JIT (CIL)
- Legacy target language: C#
Python Implementations (Unladen-Swallow benchmarks)

- Unladen-swallow
- PyPy
- IronPython
- Jython/HotSpot 7

Speedup (relative to CPython)

- nqueens
- django
- richards
- slowspifire
- float
- unpickle
- pickle
- slowpickle
- pystone
- slowunpickle
- geomean

Speed

Slowdown
Python Implementations (shootout benchmarks)
A System View of Optimizing DSL Compilers

<table>
<thead>
<tr>
<th>Lowering of IR</th>
<th>Optimizing DSL compiler</th>
<th>DSL program</th>
</tr>
</thead>
<tbody>
<tr>
<td>DSL semantics (dynamic typing, rich operators and built-in types, etc)</td>
<td>Optimizer</td>
<td>DSL interpreter + runtime</td>
</tr>
<tr>
<td>Traditional IR semantics (static typing, etc)</td>
<td>Optimizer</td>
<td></td>
</tr>
<tr>
<td>Backend IR semantics (register, instr, etc)</td>
<td>Optimizer</td>
<td></td>
</tr>
<tr>
<td>Machine semantics (ISA)</td>
<td></td>
<td>machine binary</td>
</tr>
</tbody>
</table>

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)
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
- Translation
- Traditional optimizer
- Backend
- LLVM (C)
- Legacy target language: C/C++

Jython

- DSL semantics
- Translation
- Traditional optimizer
- Backend
- Java JIT
- Legacy target language: Java

PyPy

- DSL semantics
- Translation
- Traditional optimizer
- Backend
- RPython JIT
- RPython is closest to Python

IronPython

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

<table>
<thead>
<tr>
<th>Unladen (naïve)</th>
<th>Jython</th>
<th>Unladen w/ feedback</th>
<th>PyPy</th>
</tr>
</thead>
<tbody>
<tr>
<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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>LLVM</th>
<th>JVM</th>
<th>RPython</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct opt effect</td>
<td>Semantic loss</td>
<td>Compact IR</td>
</tr>
<tr>
<td>LLVM IR</td>
<td>JVM IR</td>
<td>LLVM IR</td>
</tr>
<tr>
<td>RPython IR</td>
<td>RPython IR</td>
<td>better</td>
</tr>
</tbody>
</table>

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

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

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

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

- **Add-on trace JIT**
  - PyPy
  - LuaJIT
  - TraceMonkey
  - SPUR

Significant difference in JIT effectiveness across languages
- Javascript has the most effective JITs
- Ruby JITs are similar to Python’s
Concluding Remarks (Questions)

The challenges

Dynamically typed languages are slow. And we, the language design & implementation community, does not quite know how to optimize DSLs.

Questions for the compiler 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
− Language runtime needs to be redesigned to maximize optimizer’s capability to “strength reduce”
Performance of Javascript implementations

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

![Graph showing speedup of Ruby implementations](image-url)
BACK UP
Performance of LuaJIT

The graph shows the speedup of LuaJIT compared to the interpreter. The x-axis represents different benchmarks such as binarytrees, fasta, knucleotide-2, mandelbrot, nbody, p digits-4, regedxna-3, recomp, spectralnorm, and geomean. The y-axis represents the speedup (relative to the interpreter) with values ranging from 0 to 30.

The performance of LuaJIT is measured at different optimization levels, indicated by different colors:
- LuaJit2 (beta 4 -O0)
- LuaJit2 (beta 4 -O1)
- LuaJit2 (beta 4 -O2)
- LuaJit2 (beta 4 -O3)

The graph illustrates that LuaJIT performs well across various benchmarks, with the speedup varying depending on the optimization level.
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)