CSE P 501 – Compilers

Dynamic Languages
Hal Perkins
Winter 2016
References

• *An Efficient Implementation of Self, a dynamically-typed object-oriented language based on prototypes*, Chambers, Unger, Lee, OOPSLA 1989

• Earlier versions of this lecture by Vijay Menon, CSE 501 Sp09, adapted from slides by Kathleen Fisher
Dynamic Typing

JavaScript:

```javascript
function foo(a, b) {
  t1 = a.x;       // runtime field lookup
  t2 = b.y();     // runtime method lookup
  t3 = t1 + t2;   // runtime dispatch on ‘+’
  return t3;
}
```
Overview

• Self
  – 25+ year old research language
  – One of earliest JIT compilation systems
  – Pioneered techniques used today

• JavaScript
  – Self with a Java syntax
  – Much recent work to optimize
Self

- Prototype-based pure object-oriented language
- Designed by Randall Smith (Xerox PARC) and David Ungar (Stanford University)
  - Successor to Smalltalk-80
  - “Self: The power of simplicity” at OOPSLA ‘87
  - Initial implementation done at Stanford; then project shifted to Sun Microsystems Labs
  - Vehicle for implementation research
- Self 4.5 available from selflanguage.org
Design Goals

• Occam’s Razor: Conceptual economy
  — Everything is an object.
  — Everything done using messages.
  — No classes
  — No variables

• Concreteness
  — Objects should seem “real”
  — GUI to manipulate objects directly
How successful?

- Very well-designed language, but...
- Few users: not a popular success
- However, many research innovations
  - Very simple computational model
  - Enormous advances in compilation techniques
  - Influenced the design of Java compilers
Language Overview

- Dynamically typed
- Everything is an object
- All computation via message passing
- Creation and initialization done by copying example object
- Operations on objects:
  - send messages
  - add new slots
  - replace old slots
  - remove slots
Objects and Slots

Object consists of named slots.

- Data
  - Such slots return contents upon evaluation; so act like variables
- Assignment
  - Set the value of
- Method
  - Slot contains Self code
- Parent
  - References an object to inherit its slots
Messages and Methods

• When a message is sent, search the receiver object for a slot with that name
• If none found, all parents are searched
  – Runtime error if more than one parent has a slot with the same name
• If slot found, its contents are evaluated and returned
  – Runtime error if no slot found
Messages and Methods

obj  slot: our

obj x  →  3

obj print  →  print point object

obj x: 4  →  obj after setting x to 4.

clone  ...

parent*  ...

print  ...

parent*

x  3

x:  ←

UW CSE P 501 Winter 2016  X2-11
Mixing State and Behavior

parent* + add points

<table>
<thead>
<tr>
<th>parent*</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>4</td>
</tr>
<tr>
<td>y</td>
<td>17</td>
</tr>
<tr>
<td>x:</td>
<td>←</td>
</tr>
<tr>
<td>y:</td>
<td>←</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>parent*</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
</tr>
<tr>
<td>random number generator</td>
</tr>
<tr>
<td>y</td>
</tr>
<tr>
<td>y:</td>
</tr>
</tbody>
</table>
Object Creation

- To create an object, we copy an old one
- We can add new methods, override existing ones, or even remove methods

- These operations also apply to parent slots
Changing Parent Pointers

frog
- jump
- eatFly

prince
- dance
- eatCake

p
- parent⁺: ←
- name: ←
- name: Charles

p jump.
p eatFly.
p parent: prince.
p dance.
Changing Parent Pointers

frog
   jump  ...  prince
eatFly  ...

dance ...
eatCake ...

p
   parent' ←
   parent': ←
   name  Charles
   name': ←

p jump.
p eatFly.
p parent: prince.
p dance
Disadvantages of classes?

• Classes require programmers to understand a more complex model
  – To make a new kind of object, we have to create a new class first
  – To change an object, we have to change the class
  – Infinite meta-class regression

• But: Does Self require programmers to reinvent structure?
  – Common to structure Self programs with *traits*: objects that simply collect behavior for sharing
Contrast with C++

- C++
  - Restricts expressiveness to ensure efficient implementation
- Self
  - Provides unbreakable high-level model of underlying machine
  - Compiler does fancy optimizations to obtain acceptable performance
Implementation Challenges I

- Many, many slow function calls:
  - Function calls generally somewhat expensive
  - Dynamic dispatch makes message invocation even slower than typical procedure calls
  - OO programs tend to have lots of small methods
  - Everything is a message: even variable access!

“The resulting call density of pure object-oriented programs is staggering, and brings naïve implementations to their knees” [Chambers & Ungar, PLDI 89]
Implementation Challenges II

• No static type system
  — Each reference could point to any object, making it hard to find methods statically

• No class structure to enforce sharing
  — Copies of methods in every object creates lots of space overhead

√ Optimized Smalltalk-80 is roughly 10 times slower than optimized C
Optimization Strategies

- Avoid per object space requirements
- Compile, don’t interpret
- Avoid method lookup
- Inline methods wherever possible
  - Saves method call overhead
  - Enables further optimizations
Clone Families
(Objects created from same prototype

prototype

Model

cloned family

mutable

fixed

avoid per object data

implementation

map:

fixed

info

mutable

map
Dynamic Compilation

- Method is converted to byte codes when entered into the system
- Compiled to machine code when first executed
- Code stored in cache
  - if cache fills, previously compiled method flushed
- Requires entire source (byte) code to be available at runtime
Lookup Cache

- Cache of recently used methods, indexed by (receiver type, message name) pairs
- When a message is sent, compiler first consults cache
  - if found: invokes associated code
  - if absent: performs general lookup and potentially updates cache
- Berkeley Smalltalk would have been 37% slower without this optimization
Static Type Prediction

- Compiler predicts types that are unknown but likely:
  - Arithmetic operations (+, -, <, etc.) have small integers as their receivers 95% of the time in Smalltalk-80
  - ifTrue had Boolean receiver 100% of the time.
- Compiler inlines code (and test to confirm guess):

\[
x + y
\]

\[
\begin{array}{l}
\text{if type = smallInt jump to method_smallInt} \\
\text{call general_lookup}
\end{array}
\]

\[
\text{if } (b) \times \\
\text{if } (b) \times
\]

\[
\text{if } (b) \times e \text{ or } y \\
\text{if } (b) \times
\]

\[
\text{iftrue } b; \text{ iffalse } x \times
\]

Avoid method lookup
Inline Caches

- First message send from a call site:
  - general lookup routine invoked
  - call site back-patched
    - is previous method still correct?
      - yes: invoke code directly
      - no: proceed with general lookup & backpatch

- Successful about 95% of the time

- All compiled implementations of Smalltalk and Self use inline caches.
Polymorphic Inline Caches

- Typical call site has <10 distinct receiver types
  - Often can cache all receivers
- At each call site, for each new receiver, extend patch code:
  ```
  if type = rectangle jump to method_rect
  if type = circle jump to method_circle
  call general_lookup
  ```
- After some threshold, revert to simple inline cache (megamorphic site)
- Order clauses by frequency
- Inline short methods into PIC code
Customized Compilation

• Compile several copies of each method, one for each receiver type
• Within each copy:
  — Compiler knows the type of self
  — Calls through self can be statically selected and inlined
• Enables downstream optimizations
• Increases code size
Type Analysis

- Constructed by compiler by flow analysis
- Type: set of possible maps for object
  - Singleton: know map statically
  - Union/Merge: know expression has one of a fixed collection of maps
  - Unknown: know nothing about expression
- If singleton, we can inline method
- If type is small, we can insert type test and create branch for each possible receiver (type casing)
Message Splitting

- Type information above a merge point is often better
- Move message send “before” merge point:
  - duplicates code
  - improves type information
  - allows more inlining
PICS as Type Source

• Polymorphic inline caches build a call-site specific type database \textit{as the program runs}

• Compiler can use this runtime information rather than the result of a static flow analysis to build \textit{type cases}

• Must wait until PIC has collected information
  – When to recompile?
  – What should be recompiled?

• Initial fast compile yielding slow code; then dynamically recompile – \textit{hotspots}
Performance Improvements

- Initial version of Self was 4-5 times slower than optimized C
- Adding type analysis and message splitting got within a factor of 2 of optimized C
- Replacing type analysis with PICS improved performance by further 37%

Current Self compiler is within a factor of 2 of optimized C.
Impact on Java

Self with PICs → Sun cancels Self → Animorphics Smalltalk

✓ Java becomes popular

Animorphics Java → Sun buys A.J. → Java Hotspot
Summary of Self

• “Power of simplicity”
  — Everything is an object: no classes, no variables
  — Provides high-level model that can’t be violated (even during debugging)

• Fancy optimizations recover reasonable performance

• Many techniques now used in Java compilers

• Papers describing various optimization techniques available from Self web site
JavaScript

• Self-like language with Java syntax
  — Dynamic OO language
  — Prototypes instead of classes
  — Nothing to do with Java beyond syntax

• Originated in Netscape

• “Standard” on today’s browsers
High-performance JavaScript

• Self approach:
  — V8 (Google Chrome)
  — SquirrelFish Extreme (Safari / WebKit)

• Trace compilation:
  — TraceMonkey (Firefox)
  — Tamarin (Adobe Flash/Flex)
V8 (Google Chrome)

- Three primary features
  - Fast property access
    - Hidden classes
  - Dynamic compiler
    - Compile on first invocation
    - Inline caching with back patching
  - Generational garbage collection
    - Segmented by types
- See
  [http://code.google.com/apis/v8/design.html](http://code.google.com/apis/v8/design.html)
Trace-Based Compilation

- Interpret initially
- Record trace information
  - Single entry, multiple exit
  - Loop header is typically trace start
- Compile hot trace (hot path through flowgraph)
  - Interpreter jumps to trace code when available
  - Stitch multiple traces together
- Specialize hot path (omit redundant checks)
  - Claim this achieves benefits of inline caching