

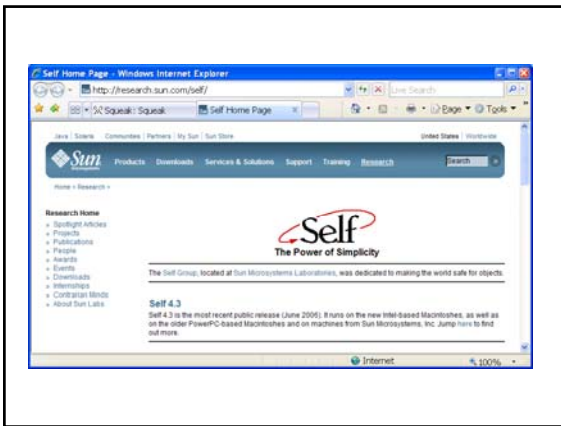
Self

John Mitchell

Slides developed by Kathleen Fisher

History

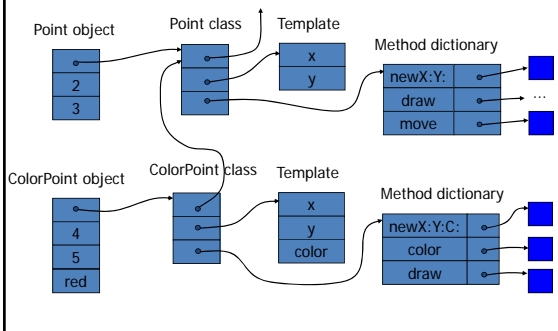
- Prototype-based pure object-oriented language.
- Designed by Randall Smith (Xerox PARC) and David Ungar (Stanford University).
 - Successor to Smalltalk-80.
 - “Self: The power of simplicity” appeared at OOPSLA ‘87.
 - Initial implementation done at Stanford; then project shifted to Sun Microsystems Labs.
 - Vehicle for implementation research.
- Self 4.2 available from Sun web site: <http://research.sun.com/self/>



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

“A Language for Smalltalk runtime structures”



How successful?

- Self is a carefully designed language
- Few users: not a popular success
 - No compelling application, until JavaScript
 - Influenced development of object calculi w/o classes
- 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

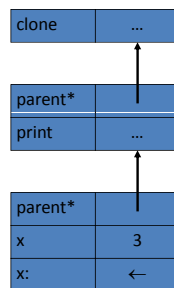
Objects and Slots

Object consists of named slots.

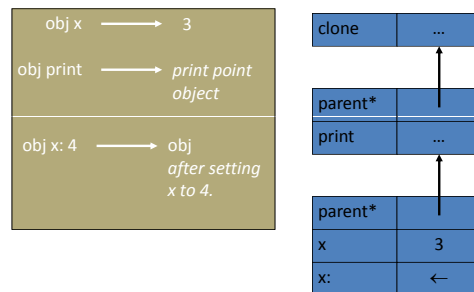
- Data
 - Such slots return contents upon evaluation; so act like instance variables
- Assignment
 - Set the value of associated slot
- Method
 - Slot contains Self code
- Parent
 - Point to existing object to inherit slots

Messages and Methods

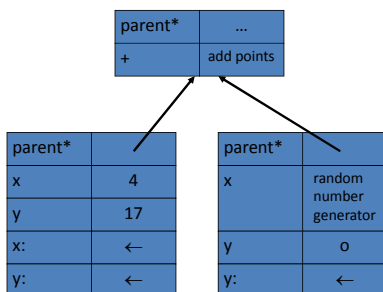
- When message is sent, object searched for slot with name.
- If none found, all parents are searched.
 - Runtime error if more than one parent has a slot with the same name.
- If slot is found, its contents evaluated and returned.
 - Runtime error if no slot found.



Messages and Methods

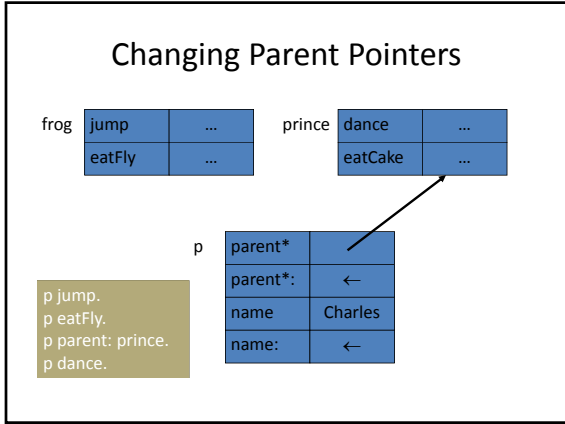
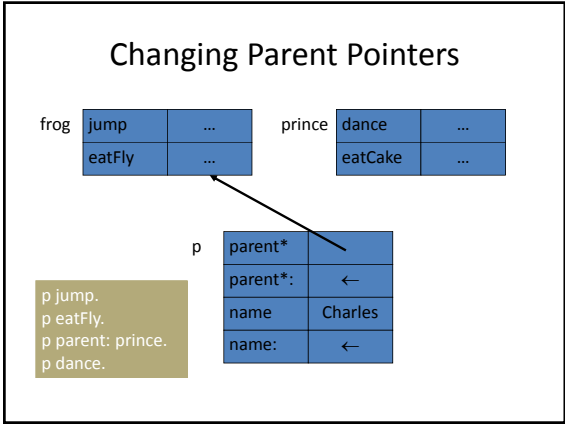


Mixing State and Behavior



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



- ### 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 programmer 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

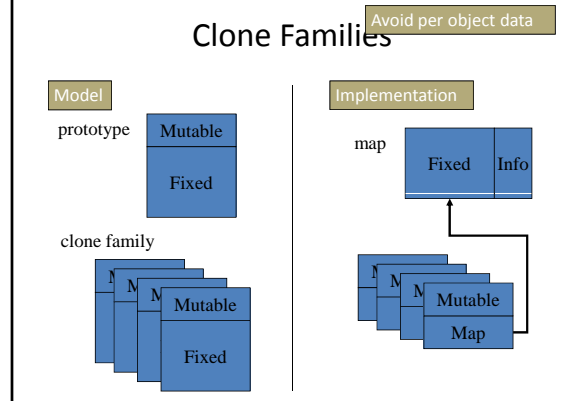
- ### Implementation Challenges I
- Many, many slow function calls:
 - Function calls generally somewhat expensive.
 - Dynamic dispatch makes message invocation even slower than typical procedure calls.
 - OO programs tend to have lots of small methods.
 - Everything is a message: even variable access!
- "The resulting call density of pure object-oriented programs is staggering, and brings naive 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
 - Each object having a copy of its methods leads to space overheads.
- Optimized Smalltalk-80 roughly 10 times slower than optimized C

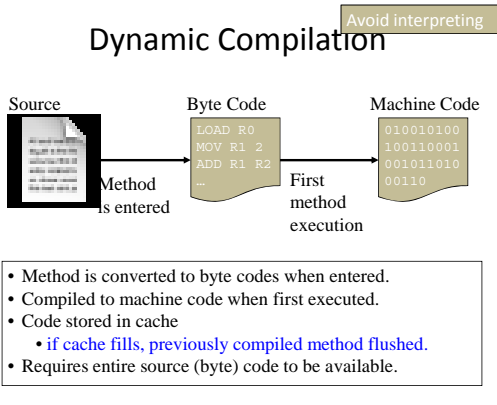
Optimization Strategies

- 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



Dynamic Compilation



Lookup Cache

- Avoid method lookup
- 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

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

```
if type = smallInt jump to method_smallInt
call_general_lookup
```

Inline Caches

- Avoid method lookup
- 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.

Avoid method lookup

Polymorphic Inline Caches

- Typical call site has <10 distinct receiver types.
 - So often can cache *all* receivers.
- At each call site, for each new receiver, extend path


```

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](#)).

Inline methods

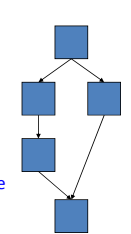
Customized Compilation

- 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.

Inline methods

Type Analysis

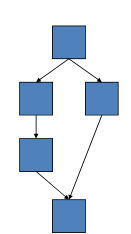
- Constructed by compiler by flow analysis.
- Type: set of possible maps for object.
 - Singleton: know map statically
 - Union/Merge: know expression has one of a fixed collection of maps.
 - Unknown: know nothing about expression.
- If singleton, we can inline method.
- If type is small, we can insert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



Inline methods

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



Inline methods

PICS as Type Source

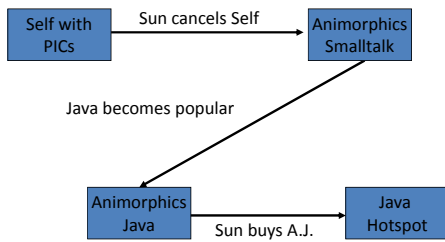
- Polymorphic inline caches build a call-site specific type database *as the program runs*.
- Compiler can use this runtime information rather than the result of a static flow analysis to build type cases.
- Must wait until PIC has collected information.
 - When to recompile?
 - What should be recompiled?
- Initial fast compile yielding slow code; then dynamically recompile *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



Summary

- “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.