Dynamic Languages

CSE 501

Spring 15

With materials adopted from John Mitchell

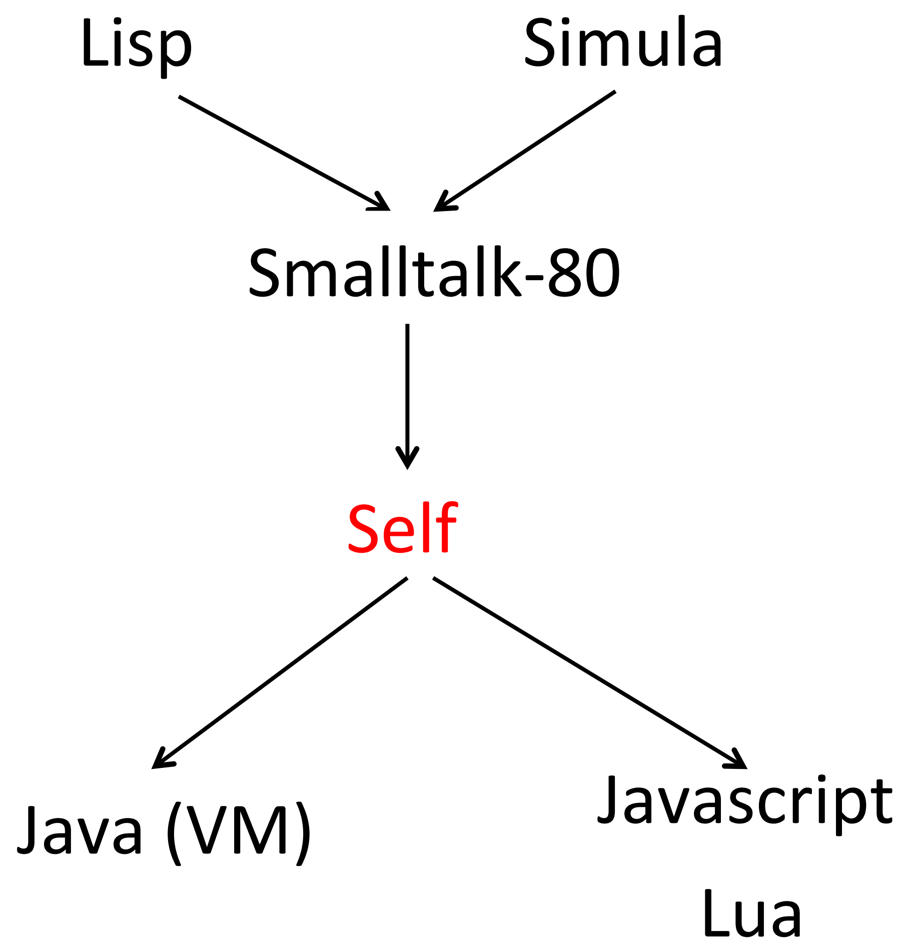
Dynamic Programming Languages

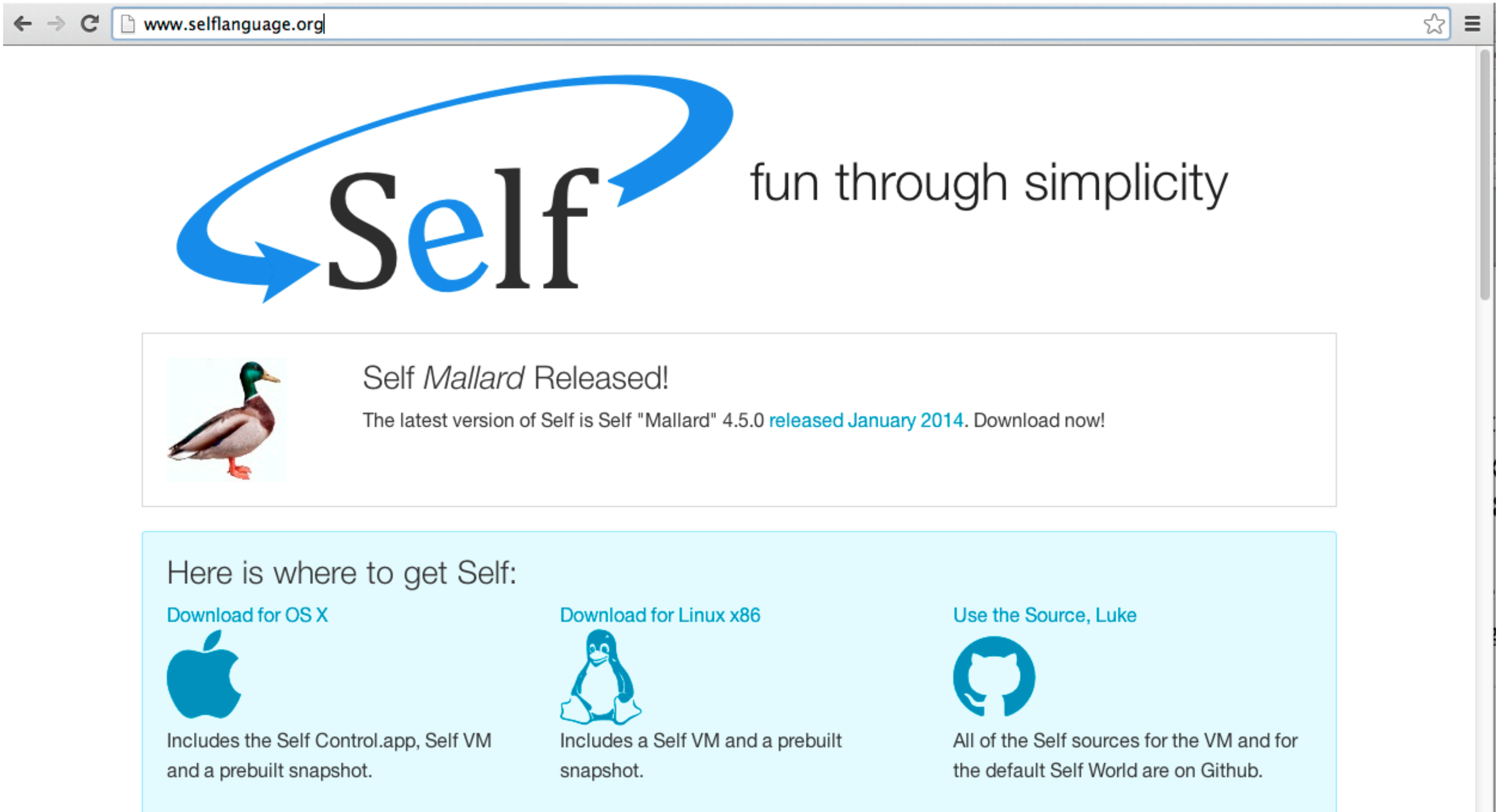
- Languages where program behavior, broadly construed, cannot be determined during compilation
  - Types
  - Code to be executed (`eval` in Javascript)
  - Loading external libraries
- Language examples
  - Javascript
  - Python
  - PHP
  - Smalltalk
  - Matlab

History of Self

- Prototype-based pure object-oriented language.
- Designed by Randall Smith (Xerox PARC) and David Ungar (Stanford) in 1987.
  - Successor to Smalltalk-80
  - Vehicle for implementation research
  - Later implementation by Craig Chambers and others at Stanford ← This is the one we are studying

History of SELF




www.selflanguage.org
Self
fun through simplicity
Self *Mallard* Released!
The latest version of Self is Self "Mallard" 4.5.0 released January 2014. Download now!
Here is where to get Self:
Download for OS X
Includes the Self Control.app, Self VM and a prebuilt snapshot.
Download for Linux x86
Includes a Self VM and a prebuilt snapshot.
Use the Source, Luke
All of the Self sources for the VM and for the default Self World are on Github.

Design Goals

- Conceptual economy
  - Everything is an object
  - Everything done using messages
  - No classes
  - No variables
- Concreteness
  - Objects should seem “real”
  - GUI to manipulate objects directly

Language Overview

- Dynamically typed
  - Users do not declare types
- All computation via message passing
- Objects are organized into slots
- Operations on objects:
  - send messages
  - add new slots
  - replace old slots
  - remove slots

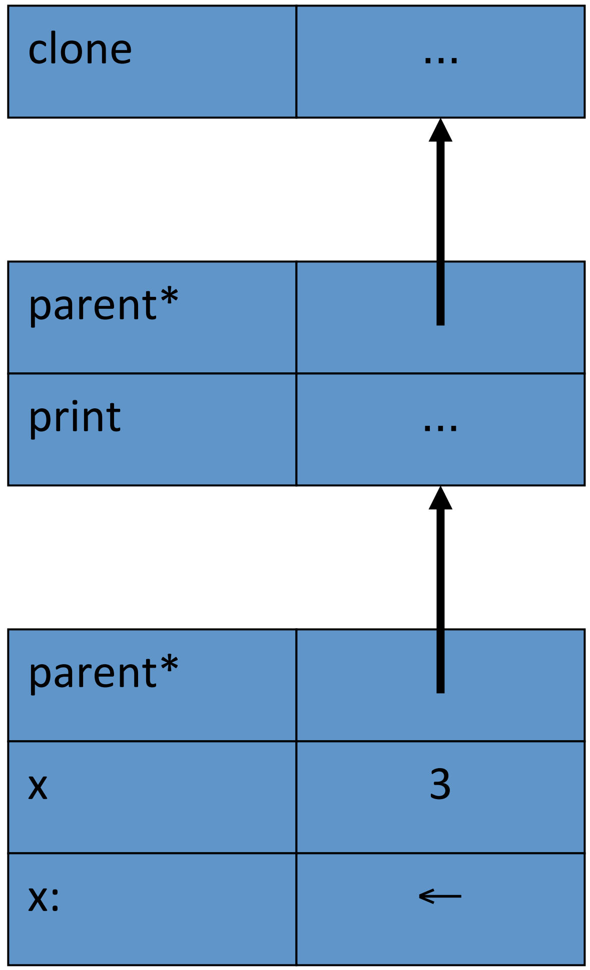
Objects and Slots

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 otherwise

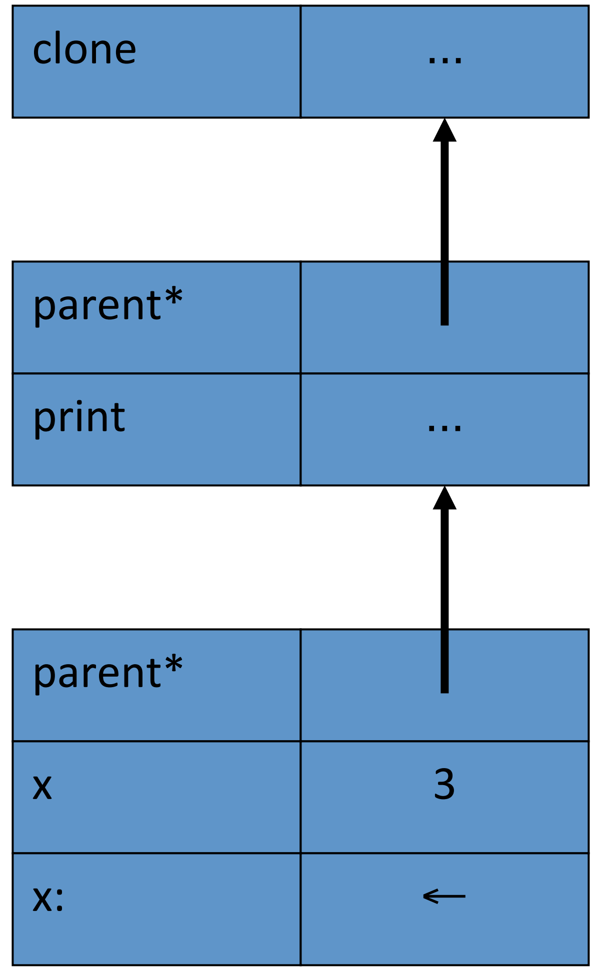


Messages and Methods

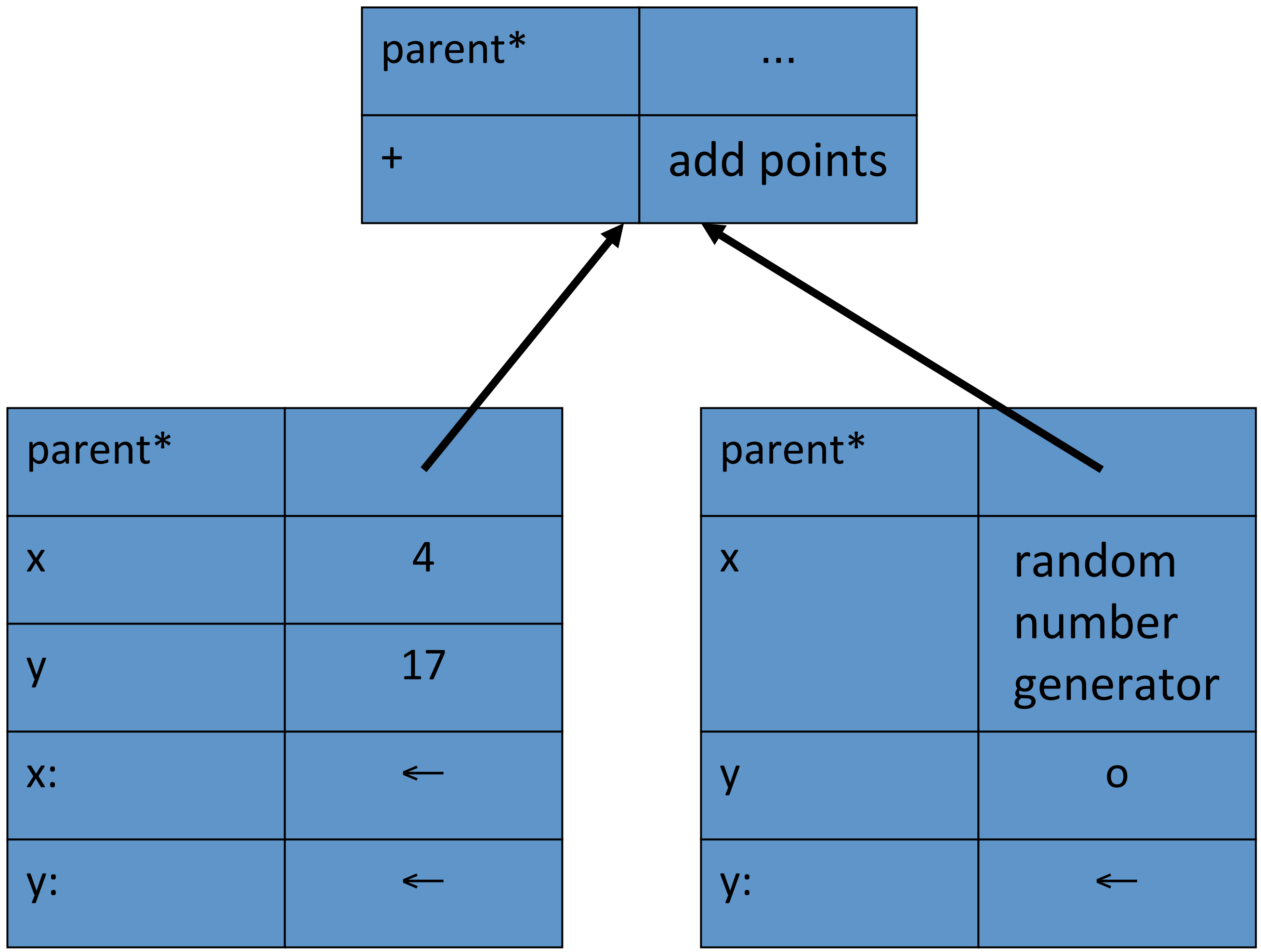
obj x → 3

obj print → *print point object*

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



Mixing State and Behavior

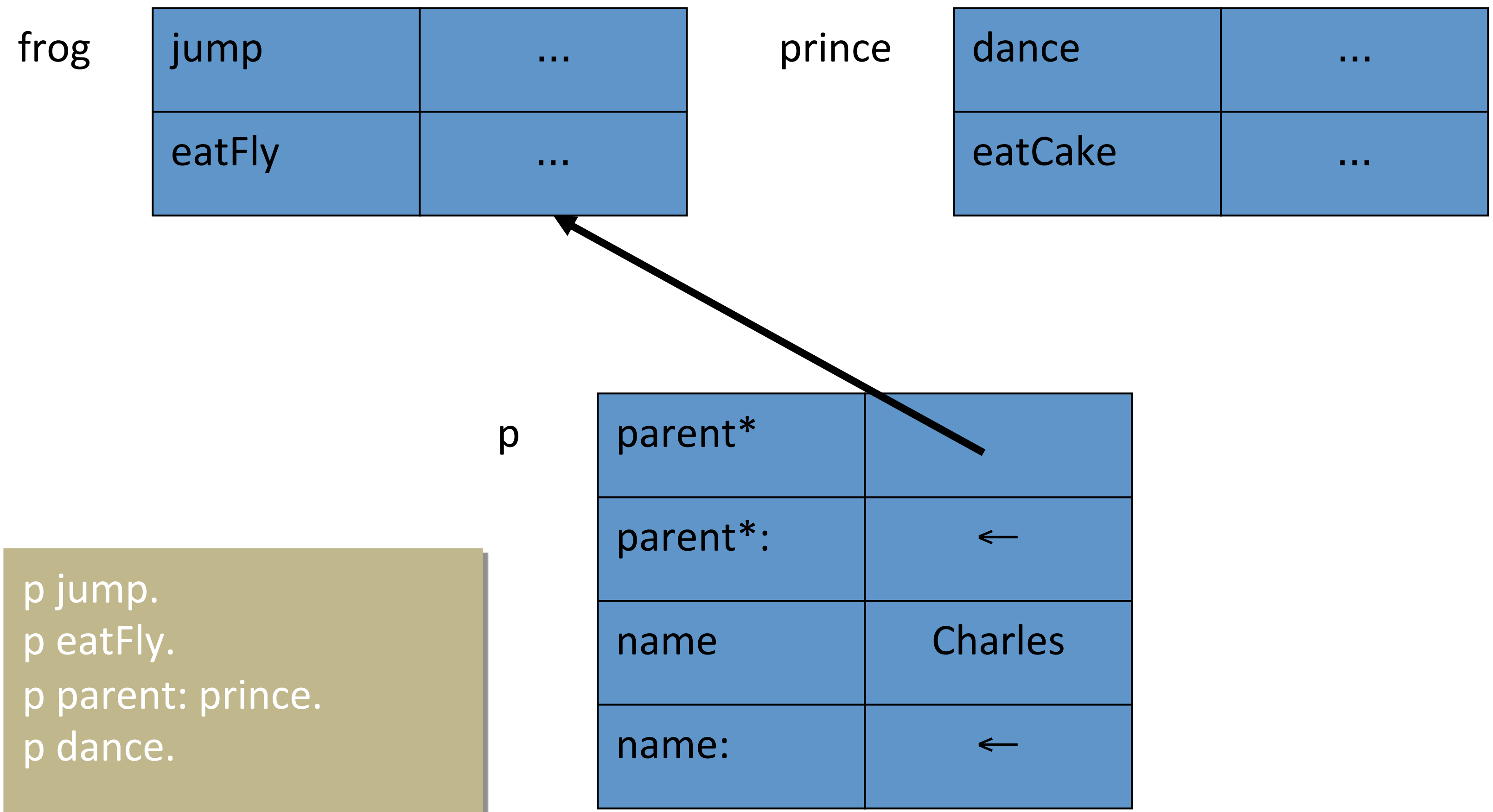


Creating Objects

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

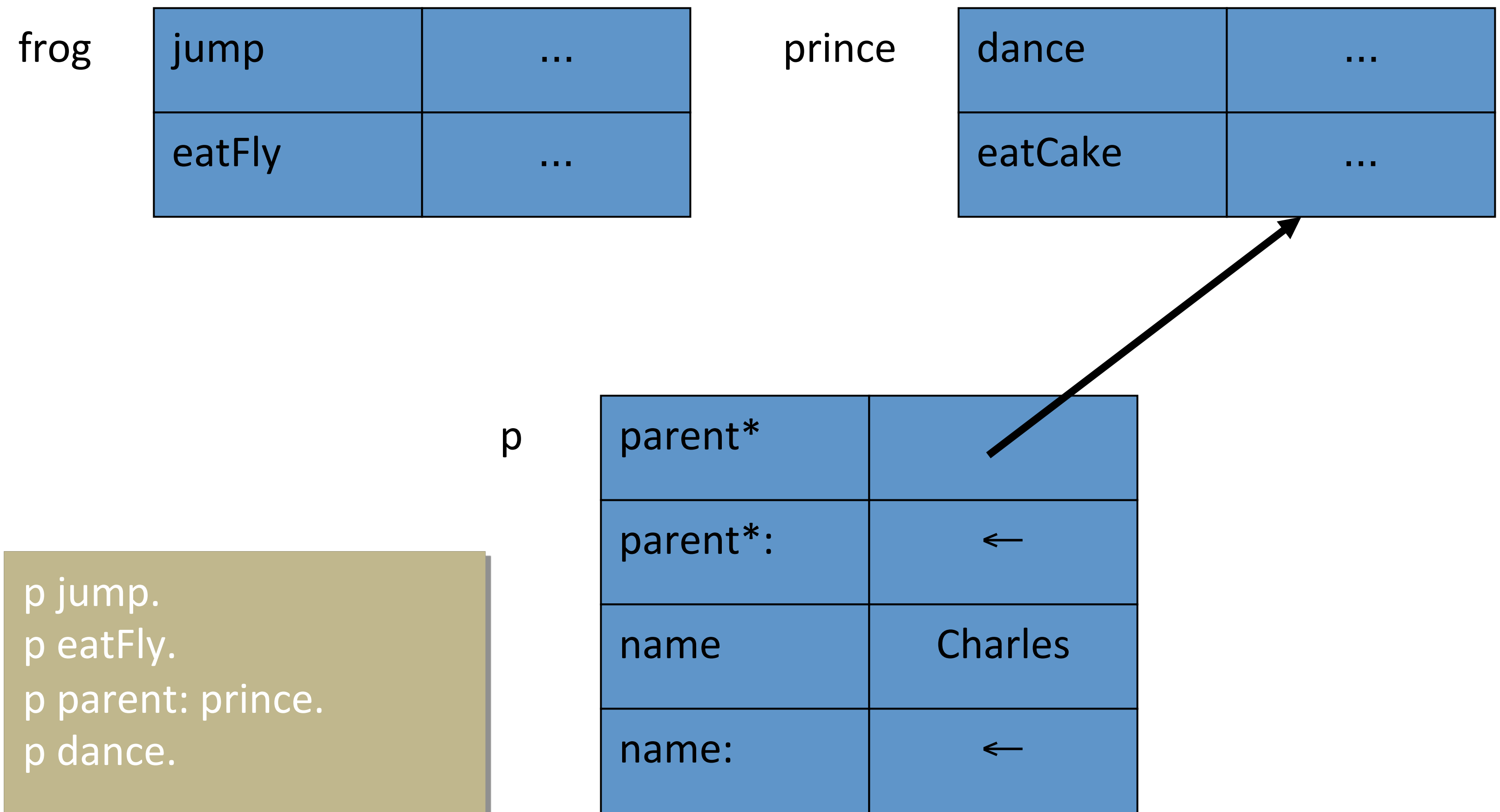
- These operations also apply to parent slots as well

Changing Parent Pointers




```
p jump.
p eatFly.
p parent: prince.
p dance.
```

Changing Parent Pointers




```
p jump.
p eatFly.
p parent: prince.
p dance.
```

Why no 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++

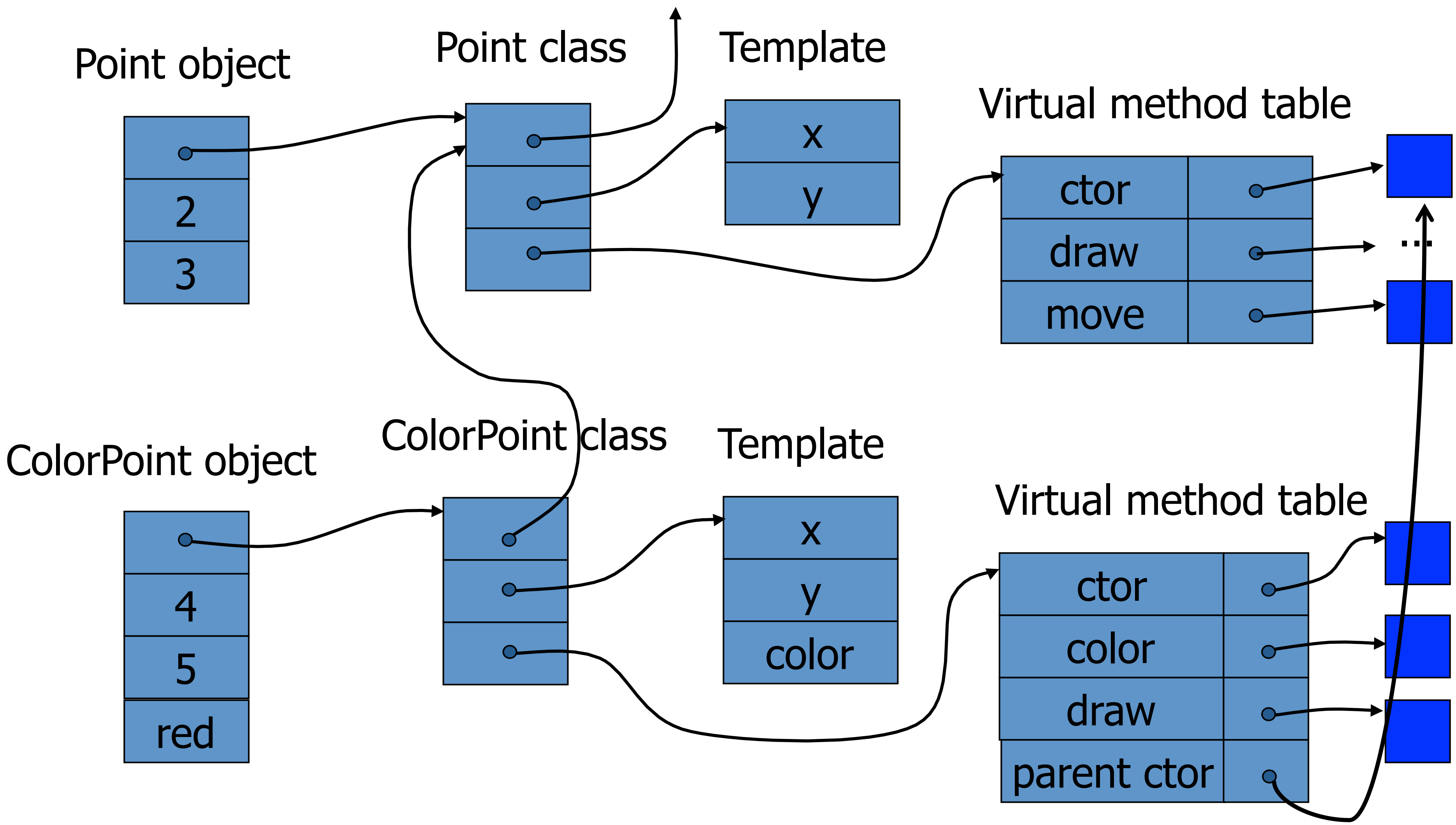
- C++
  - Restricts expressiveness to ensure efficient implementation
    - Class hierarchy is fixed during development
- Self
  - Provides high-level abstraction of underlying machine
  - Compiler does fancy optimizations to obtain acceptable performance

Implementation Challenges I

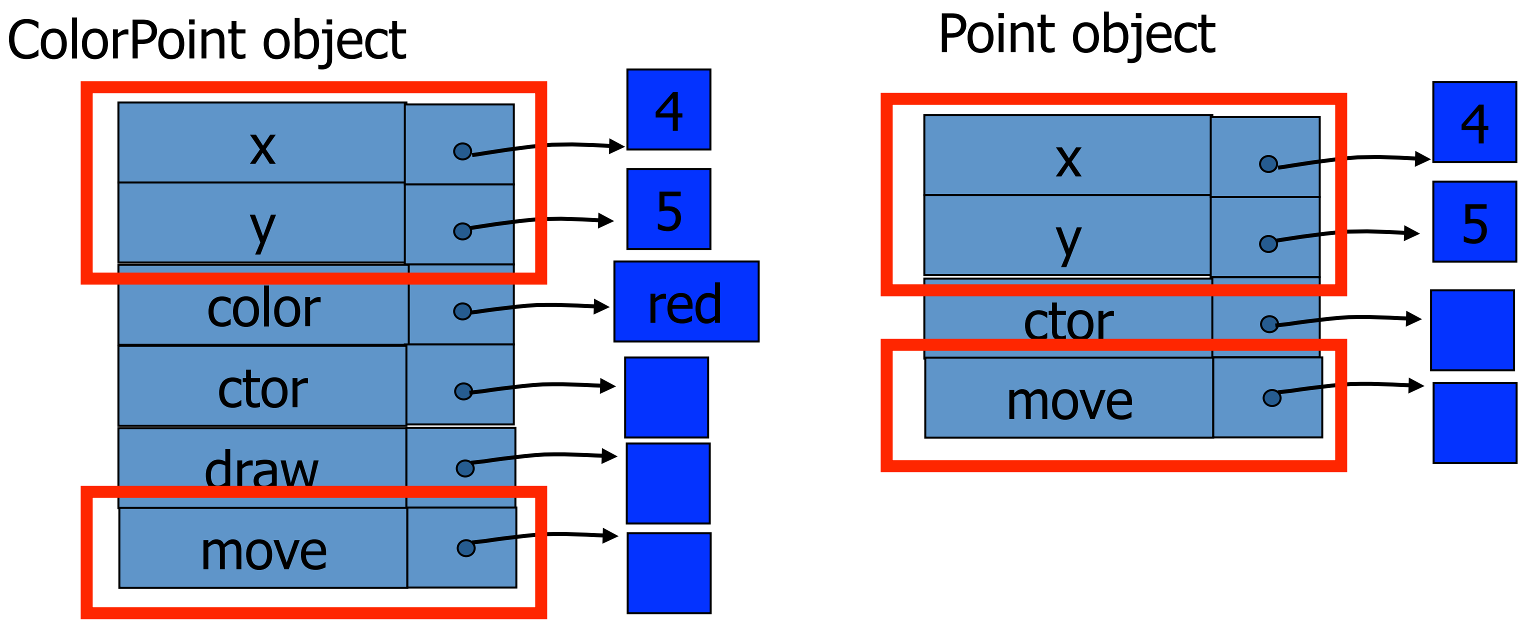
- Many, many slow function calls:
  - Function calls generally expensive.
  - Dynamic dispatch makes message invocation even slower than typical procedure calls.
  - OO programs tend to have lots of small methods.
  - Everything is a message: even variable access!

> "The resulting call density of pure object-oriented programs is staggering, and brings naïve implementations to their knees"
>
> [Chambers & Ungar, PLDI 89]

C++ Object Layout



Naive Self Object Layout



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.

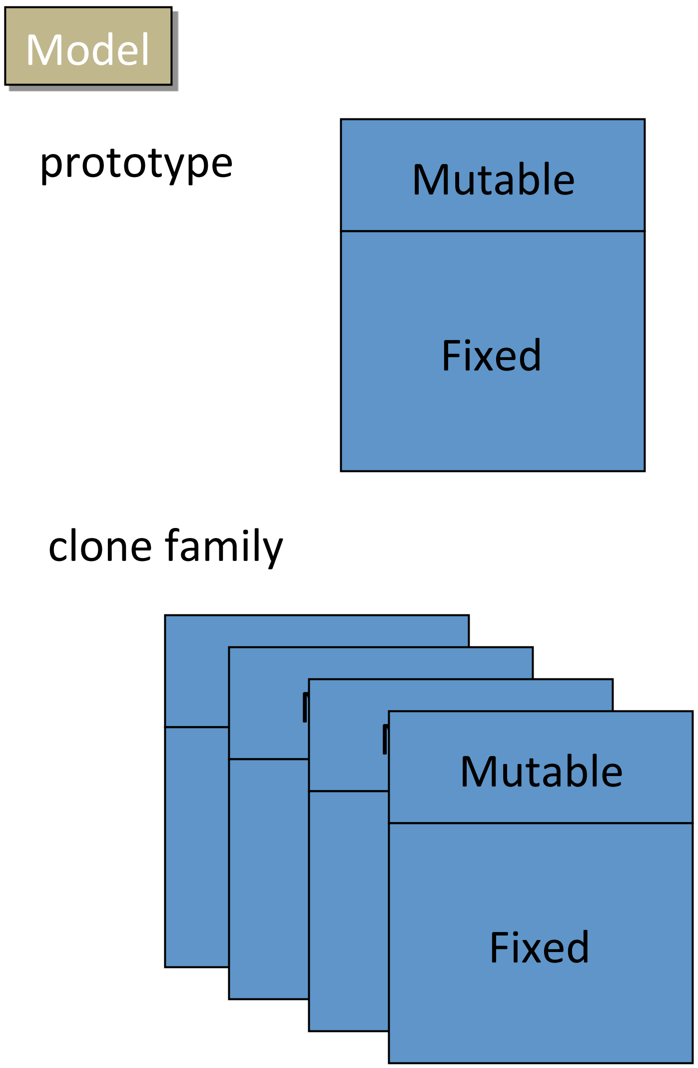
Optimized Smalltalk-80 roughly 10 times slower than optimized C

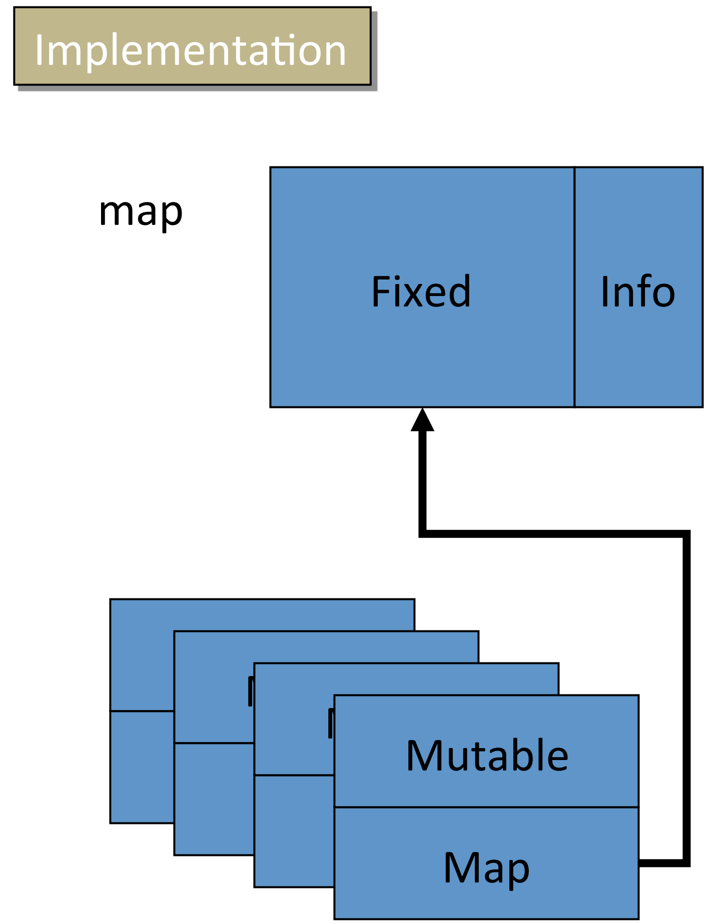
Optimization Strategies

- Avoid per object space requirements
- Avoid interpreting
  - Compile code instead
- Avoid method lookup
- Inline methods wherever possible
  - Saves method call overhead
  - Enables further optimizations

Clone Families

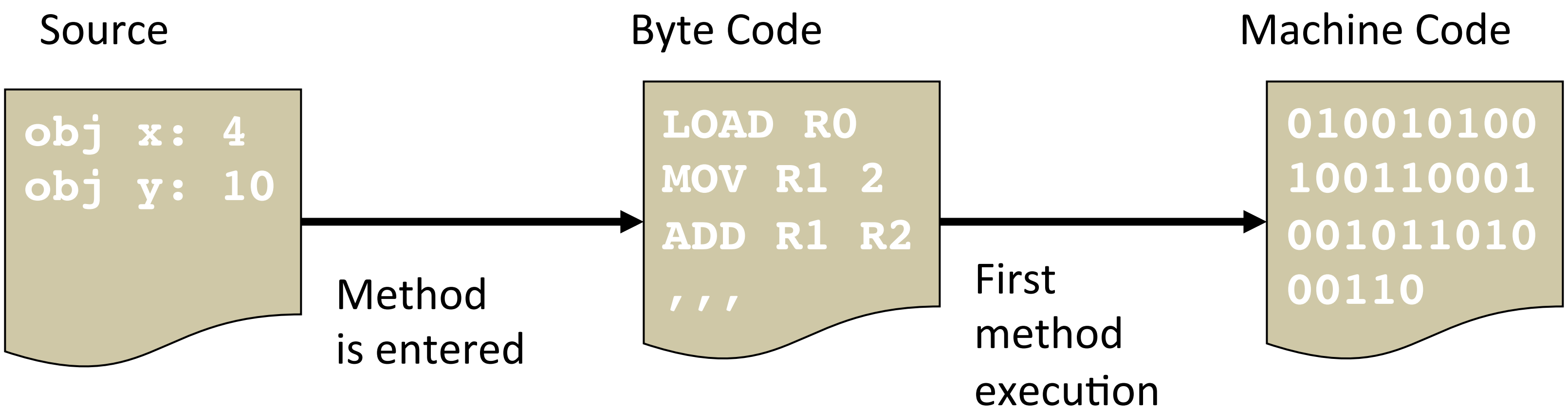
Avoid per object data





Dynamic Compilation

Avoid interpreting



- Method is converted to byte codes when entered
- 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

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

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
else 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 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

Customized Compilation

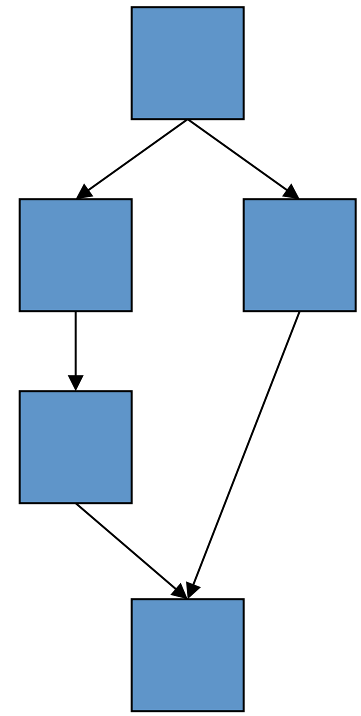
Inline methods

- 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

Inline methods

- 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)



Performance Improvements

- Initial version of Self was 4-5 times slower than optimized C.
- After optimizations, implementation described in paper is within a factor of 2 of optimized C.

How successful?

- 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
  - Direct influence on design of Javascript

Lessons

Pochoir / Halide (DSL)

- Design specific constructs for domain
- Constructs need to easily map to underlying target language
  - Otherwise implementation might be a nightmare
- Expose high-level structure allows domain-specific optimizations

Self / Javascript (Dynamic Languages)

- "Power of simplicity"
  - Everything is an object
  - No classes, no variables
- Implementation specific to program constructs
- Uses various optimization tricks to recover performance