Prototype-based languages

CS565

Purdue University

March 23rd, 2010

Type systems

As with all things in languages, type systems range from static to dynamic.

- Static classes: All types are known (and often specified) at compile time. Types cannot change, so methods and fields of objects are known.
- Dynamic classes: Classes are a special kind of object. e.g. Python. Types are generally known, and often information can be inferred as if they were static classes.
- **Prototypes**: No classes, prototypes are *not* a special kind of object. e.g. JavaScript. Since prototypes are not a special kind of object (the name "prototype" is bestowed by programmers, not the language), all bets are off.

(Note that this summary ignores the distinction between strongly and weakly typed type systems)

Prototypes

Should be review: objects have a prototype, field lookup involves looking through prototypes.

Self

- Self is the language that invented prototype-based programming
- Introduced in *Self: The power of simplicity* (David Ungar and Randall B. Smith)
- Similar to Smalltalk in many ways (which was the big academic OO language of the time)

Self syntax

```
object unaryMethod
object + object
object nary: arg1 method: arg2
```

(The methods above are "unaryMethod", "+" and "nary:method:")

Self syntax — adding slots

Self, like Smalltalk, is intrinsically tied to a graphical environment; adding slots is not generally done with code. But it can be:

```
object _AddSlots: (| foo = (a method).
                     bar <- (a field) |)
```

Data fields are hidden by two (overwritable) accessor methods:

```
object _AddSlots: (| innerfoo <- (a real field).
                     foo = (getter).
                     foo: = (setter) |)
```

Self prototypes

Self introduced prototypes, but the features of Self prototypes are quite different from e.g. JavaScript prototypes.

- The fundamental behavior is to *copy* objects

  ```
  mycar <- car copy
  ```

- May also explicitly state inheritance relationships by creating prototype fields:

  ```
  parent* <- protoobject
  ```

- Any number of prototype fields may be created:

  ```
  parent1* <- a.
  parent2* <- b
  ```

- Copies have their own data, children do not

Self inheritance

```
lobby _AddSlots: (| foo <- (| parent* <- traits clonable.
                              x <- 0.
                              setx: val = (x: val) |) |)
lobby _AddSlots: (| foochild <- (| parent* <- foo |).
                    foocopy <- foo copy.
                    foochild2 <- (| parent* <- foo.
                                    x <- 0. |) |).


foo x "0"
foochild x "0"

foochild x: 1
foo x "1"
foochild x "1"
foocopy x "0"
foochild2 x "0"
```

Self inheritance — traits

- As a result of this means of inheritance, it is common to separate behavior and state
- Behavior = traits, inherited
- State = object, copied

Lessons learned from Self

- Encouraging copying causes people to separate state from behavior
- But behavior requires certain fields, so objects linked implicitly
- So is all this really any different from classes?

Not classes!

Yes! The interpreter knows nothing of traits.

JavaScript (and how it is not Self)

JavaScript's object system is fundamentally similar to Self. Except,

- No copying[1]
- No multiple inheritance
- No explicit setting of prototype object
- Slots are fields which may contain functions, instead of methods that may hide data
- "this" ("self" in Self) is dynamic

[1]Manual copying is possible[2], but not encouraged
[2]Except when it's not

JavaScript (and why it is not Self)

Certainly Self is more flexible. So why is JavaScript different?

- Common idioms of Self are trait+object, factory methods, both are easily rolled into constructor+prototype
- Multiple inheritance always[3] causes confusion
- Objects rarely (OK, never) change prototype in Self in practice

[3] Often?

JavaScript (and how it is Self)

For all its differences from Self, JavaScript has fundamental similarities

- The interpreter knows nothing about types (except primitives)
- Field lookup is recursive through multiple objects
- Implementation is (relatively!) easy to do, but tough to do fast

A defense of prototypes

- A prototype-based object system is difficult to make fast
- Static classes can be really fast (see C++)
- But these are dynamic languages!

Static classes

- Static classes are fast because they disappear during compilation
- At runtime, calling a method is a pointer dereference
- Difficult[4] to accomplish with prototypes

[4] But not impossible!

Dynamic classes

- But many dynamic languages use classes too
- These classes can be changed, so a quick pointer-dereference is insufficient
- Dynamic classes don't contain state (they model it)
- So the interpreter needs to know how to handle them separately from objects,
- but gains little speed benefit from doing so

Prototypes!

Interpreter doesn’t care, but not appreciably slower than dynamic classes

Interpreting prototype-based languages fast

Not only did Self introduce prototype-based languages, it revolutionized interpreter speed

(Techniques presented here extremely briefly, roughly in order of publication)

JIT

- Just-in-time compilation
- (Not specific to prototype-based languages)
- When a method is run a lot, compile it down to native code
- Even if the native code is naiv̈e, it will be faster than interpreted
- Optimize more with even-more-used methods

L. Peter Deutsch and Allan M. Schiffman. Efficient implementation of the Smalltalk–80 system. In Conference Record of the 11th Annual ACM Symposium on Principles of Programming Languages, Salt Lake City, pages 297–302, January 1984.

Customization

- Object literals define possible types
- Cutomize methods based on types likely to be provided as arguments
- When compiling, create type checks that call optimized versions if possible
- Not necessarily suitable for JavaScript, since fields are usually added in code by a constructor, not a relatively-clear object literal

Customization: Optimizing Compiler Technology for Self, a Dynamically-Typed Object-Oriented Programming Language. Proceedings of the SIGPLAN 89 Conference on Programming Language Design and Implementation, pp. 146-160, Portland, OR, June, 1989.

Maps[5]

- Most objects are created by copying
- So when objects are copied, separate out a "map" of slot names to offsets
- When objects are changed in an incompatible way, create a new map
- Customize methods per receiver map

An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes.

OOPSLA 89 Conference Proceedings, pp. 49-70, New Orleans, LA, October, 1989.

[4] v8 calls these "hidden classes", which probably fits programmers' assumptions, and that terminology is hinted at by the original paper

Abstract interpretation

- Abstract interpretation is (extremely briefly) interpreting the code using only types (not data), and relaxing the types as necessary, until all types are known (although they might be too broad, e.g. everything reduces to Object)
- Similar techniques can be used to make type checks only once, early

Patrick Cousot, Radhia Cousot: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. POPL 1977: 238-252 Iterative Type Analysis and Extended Message Splitting: Optimizing Dynamically-Typed Object-Oriented Programs. Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation, pp. 150-164, White Plains, NY, June, 1990. There are hundreds of other relevant papers, this is only the first one on the subject and a paper that applies a similar technique to Self

Inline caching

- Observation: Generally, any call site will always call the same method
- Even if it calls more than one, it's likely to call only a small number
- So, store the method called at the call site
- Simpler check on the object (type check instead of lookup)
- However, need to be able to quickly check methods may be overridden
- (JavaScript) So, consider "types" to be different if parent fields are overridden

An Efficient Implementation of Self, a Dynamically-Typed Object-Oriented Language Based on Prototypes. OOPSLA 89 Conference Proceedings, pp. 49-70, New Orleans, LA, October, 1989. Optimizing Dynamically-Typed Object-Oriented Programming Languages with Polymorphic Inline Caches. ECOOP 91 Conference Proceedings, Geneva, Switzerland, July, 1991.

And JavaScript?

Self made many advances in interpreter speed, but how many of them apply to JavaScript? The problems:

- JavaScript makes it much easier to add fields to objects (and makes it almost necessary in constructors)
- Object descriptions aren't so neat and tidy, object literals are rarer than iterative creation
- The distinction between methods and fields is fuzzy (nonexistent, really)

Other prototype-based languages

Although Self (the first) and JavaScript (the most popular) are the major ones to know, there are others:

- Io (sort of a spiritual successor to Self)
- Lisaac (crazy-fast, statically typed, compiles to relatively-efficient C)
- Lua (popular in games, fundamentally similar to JavaScript)