Self (programming language)

Self is an object-oriented programming language based on the concept of *prototypes*. Self began as a dialect of Smalltalk, being dynamically typed and using just-in-time compilation (JIT) as well as the prototype-based approach to objects: it was first used as an experimental test system for language design in the 1980s and 1990s. In 2006, Self was still being developed as part of the Klein project, which was a Self virtual machine written fully in Self. The latest version is 2017.1 released in May 2017.[2]

Several just-in-time compilation techniques were pioneered and improved in Self research as they were required to allow a very high level object oriented language to perform at up to half the speed of optimized C. Much of the development of Self took place at Sun Microsystems, and the techniques they developed were later deployed for Java's HotSpot virtual machine.

At one point a version of Smalltalk was implemented in Self. Because it was able to use the JIT, this also gave extremely good performance.[3]

Self

Paradigm	object-oriented (prototype-based)
Designed by	David Ungar, Randall Smith
Developer	David Ungar, Randall Smith, Stanford University, Sun Microsystems
First appeared	1987
Stable release	Mandarin 2017.1 / May 24, 2017
Typing discipline	dynamic, strong
License	BSD-like license
Website	www.selflanguage.org (http://www.selflanguage.org)
Major implementations	
Self	
Influenced by	
Smalltalk, APL[1]	
Influenced	
NewtonScript, JavaScript, Io, Agora, Squeak, Lua, Factor, REBOL	

Contents

History

Self was designed mostly by David Ungar and Randall Smith in 1986 while working at Xerox PARC. Their objective was to push forward the state of the art in object-oriented programming language research, once Smalltalk-80 was released by the labs and began to be taken seriously by the industry. They moved to Stanford University and continued work on the language, building the first working Self compiler in 1987. At that point, focus changed to attempting to bring up an entire system for Self, as opposed to just the language.

The first public release was in 1990, and the next year the team moved to Sun Microsystems where they continued work on the language. Several new releases followed until falling largely dormant in 1995 with the 4.0 version. The 4.3 version was released in 2006 and ran on Mac OS X and Solaris. A new release in 2010,[4] version 4.4, has been developed by a group comprising some of the original team and independent programmers and is available for Mac OS X and Linux, as are all following versions. The follow-up 4.5 was released in January 2014,[5] and three years later, version 2017.1 was released in May 2017.

Self also inspired a number of languages based on its concepts. Most notable, perhaps, were NewtonScript for the Apple Newton and JavaScript used in all modern browsers. Other examples include Io, Lisaac and Agora. The IBM Tivoli Framework's distributed object system, developed in 1990, was, at the lowest level, a prototype based object system inspired by Self.

Prototype-based programming languages

Traditional class-based OO languages are based on a deep-rooted duality:

1. Classes define the basic qualities and behaviours of objects.
2. Object instances are particular manifestations of a class.

For example, suppose objects of the `Vehicle` class have a *name* and the ability to perform various actions, such as *drive to work* and *deliver construction materials*. `Bob's car` is a particular object (instance) of the class `Vehicle`, with the name "Bob's car". In theory one can then send a message to `Bob's car`, telling it to *deliver construction materials*.

This example shows one of the problems with this approach: Bob's car, which happens to be a sports car, is not able to carry and deliver construction materials (in any meaningful sense), but this is a capability that `Vehicle`s are modelled to have. A more useful model arises from the use of subclassing to create specializations of `Vehicle`; for example `Sports Car` and `Flatbed Truck`. Only objects of the class `Flatbed Truck` need provide a mechanism to *deliver construction materials*; sports cars, which are ill-suited to that sort of work, need only *drive fast*. However, this deeper model requires more insight during design, insight that may only come to light as problems arise.

This issue is one of the motivating factors behind **prototypes**. Unless one can predict with certainty what qualities a set of objects and classes will have in the distant future, one cannot design a class hierarchy properly. All too often the program would eventually need added behaviours, and sections of the system would need to be re-designed (or refactored) to break out the objects in a different way. Experience with early OO languages like Smalltalk showed that this sort of issue came up again and again. Systems would tend to grow to a point and then become very rigid, as the basic classes deep below the programmer's code grew to be simply "wrong". Without some way to easily change the original class, serious problems could arise.

Dynamic languages such as Smalltalk allowed for this sort of change via well-known methods in the classes; by changing the class, the objects based on it would change their behaviour. However, such changes had to be done very carefully, as other objects based on the same class might be expecting this "wrong" behavior: "wrong" is often dependent on the context. (This is one form of the fragile base class

problem.) Further, in languages like C++, where subclasses can be compiled separately from superclasses, a change to a superclass can actually break precompiled subclass methods. (This is another form of the fragile base class problem, and also one form of the fragile binary interface problem.)

In Self, and other prototype-based languages, the duality between classes and object instances is eliminated.

Instead of having an "instance" of an object that is based on some "class", in Self one makes a copy of an existing object, and changes it. So `Bob's car` would be created by making a copy of an existing "Vehicle" object, and then adding the *drive fast* method, modelling the fact that it happens to be a Porsche 911. Basic objects that are used primarily to make copies are known as *prototypes*. This technique is claimed to greatly simplify dynamism. If an existing object (or set of objects) proves to be an inadequate model, a programmer may simply create a modified object with the correct behavior, and use that instead. Code which uses the existing objects is not changed.

Description

Self objects are a collection of "slots". Slots are accessor methods that return values, and placing a colon after the name of a slot sets the value. For example, for a slot called "name",

```
myPerson name
```

returns the value in name, and

```
myPerson name:'foo'
```

sets it.

Self, like Smalltalk, uses *blocks* for flow control and other duties. Methods are objects containing code in addition to slots (which they use for arguments and temporary values), and can be placed in a Self slot just like any other object: a number for example. The syntax remains the same in either case.

Note that there is no distinction in Self between fields and methods: everything is a slot. Since accessing slots via messages forms the majority of the syntax in Self, many messages are sent to "self", and the "self" can be left off (hence the name).

Basic syntax

The syntax for accessing slots is similar to that of Smalltalk. Three kinds of messages are available:

unary
    receiver `slot_name`
binary
    receiver `+` *argument*
keyword
    receiver `keyword:` *arg1* `With:` *arg2*

All messages return results, so the receiver (if present) and arguments can be themselves the result of other messages. Following a message by a period means Self will discard the returned value. For example:

```
'Hello, World!' print.
```

This is the Self version of the hello world program. The ' syntax indicates a literal string object. Other literals include numbers, blocks and general objects.

Grouping can be forced by using parentheses. In the absence of explicit grouping, the unary messages are considered to have the highest precedence followed by binary (grouping left to right) and the keywords having the lowest. The use of keywords for assignment would lead to some extra parenthesis where expressions also had keyword messages, so to avoid that Self requires that the first part of a keyword message selector start with a lowercase letter, and subsequent parts start with an uppercase letter.

```
valid: base bottom
          between: ligature bottom + height
          And: base top / scale factor.
```

can be parsed unambiguously, and means the same as:

```
valid: ((base bottom)
            between: ((ligature bottom) + height)
            And: ((base top) / (scale factor))).
```

In Smalltalk-80, the same expression would look written as:

```
valid := self base bottom
             between: self ligature bottom + self height
             and: self base top / self scale factor.
```

assuming `base`, `ligature`, `height` and `scale` were not instance variables of `self` but were, in fact, methods.

Making new objects

Consider a slightly more complex example:

```
labelWidget copy label: 'Hello, World!'.
```

makes a copy of the "labelWidget" object with the copy message (no shortcut this time), then sends it a message to put "Hello, World" into the slot called "label". Now to do something with it:

```
(desktop activeWindow) draw: (labelWidget copy label: 'Hello, World!').
```

In this case the `(desktop activeWindow)` is performed first, returning the active window from the list of windows that the desktop object knows about. Next (read inner to outer, left to right) the code we examined earlier returns the labelWidget. Finally the widget is sent into the draw slot of the active window.

Delegation

In theory, every Self object is a stand-alone entity. Self has neither classes nor meta-classes. Changes to a particular object do not affect any other, but in some cases it is desirable if they did. Normally an object can understand only messages corresponding to its local slots, but by having one or more slots indicating *parent* objects, an object can **delegate** any message it does not understand itself to the parent object. Any slot can

be made a parent pointer by adding an asterisk as a suffix. In this way Self handles duties that would use inheritance in class-based languages. Delegation can also be used to implement features such as namespaces and lexical scoping.

For example, suppose an object is defined called "bank account", that is used in a simple bookkeeping application. Usually, this object would be created with the methods inside, perhaps "deposit" and "withdraw", and any data slots needed by them. This is a prototype, which is only special in the way it is used since it also happens to be a fully functional bank account.

Traits

Making a clone of this object for "Bob's account" will create a new object which starts out exactly like the prototype. In this case we have copied the slots including the methods and any data. However a more common solution is to first make a more simple object called a traits object which contains the items that one would normally associate with a class.

In this example the "bank account" object would not have the deposit and withdraw method, but would have as a parent an object that did. In this way many copies of the bank account object can be made, but we can still change the behaviour of them all by changing the slots in that root object.

How is this any different from a traditional class? Well consider the meaning of:

```
myObject parent: someOtherObject.
```

This excerpt changes the "class" of myObject at runtime by changing the value associated with the 'parent*' slot (the asterisk is part of the slot name, but not the corresponding messages). Unlike with inheritance or lexical scoping, the delegate object can be modified at runtime.

Adding slots

Objects in Self can be modified to include additional slots. This can be done using the graphical programming environment, or with the primitive '_AddSlots:'. A **primitive** has the same syntax as a normal keyword message, but its name starts with the underscore character. The _AddSlots primitive should be avoided because it is a left over from early implementations. However, we will show it in the example below because it makes the code shorter.

An earlier example was about refactoring a simple class called Vehicle in order to be able to differentiate the behaviour between cars and trucks. In Self one would accomplish this with something like this:

```
_AddSlots: (| vehicle <- (|parent* = traits clonable|) |).
```

Since the receiver of the '_AddSlots:' primitive isn't indicated, it is "self". In the case of expressions typed at the prompt, that is an object called the "lobby". The argument for '_AddSlots:' is the object whose slots will be copied over to the receiver. In this case it is a literal object with exactly one slot. The slot's name is 'vehicle' and its value is another literal object. The "<-" notation implies a second slot called 'vehicle:' which can be used to change the first slot's value.

The "=" indicates a constant slot, so there is no corresponding 'parent:'. The literal object that is the initial value of 'vehicle' includes a single slot so it can understand messages related to cloning. A truly empty object, indicated as (| |) or more simply as (), cannot receive any messages at all.

```
vehicle _AddSlots: (| name <- 'automobile'|).
```

Here the receiver is the previous object, which now will include 'name' and 'name:' slots in addition to 'parent*'.

```
_AddSlots: (| sportsCar <- vehicle copy |).
sportsCar _AddSlots: (| driveToWork = (''some code, this is a method'') |).
```

Though previously 'vehicle' and 'sportsCar' were exactly alike, now the latter includes a new slot with a method that the original doesn't have. Methods can only be included in constant slots.

```
_AddSlots: (| porsche911 <- sportsCar copy |).
porsche911 name:'Bobs Porsche'.
```

The new object 'porsche911' started out exactly like 'sportsCar', but the last message changed the value of its 'name' slot. Note that both still have exactly the same slots even though one of them has a different value.

Environment

One feature of Self is that it is based on the same sort of virtual machine system that earlier Smalltalk systems used. That is, programs are not stand-alone entities as they are in languages such as C, but need their entire memory environment in order to run. This requires that applications be shipped in chunks of saved memory known as *snapshots* or *images*. One disadvantage of this approach is that images are sometimes large and unwieldy; however, debugging an image is often simpler than debugging traditional programs because the runtime state is easier to inspect and modify. (The difference between source-based and image-based development is analogous to the difference between class-based and prototypical object-oriented programming.)

In addition, the environment is tailored to the rapid and continual change of the objects in the system. Refactoring a "class" design is as simple as dragging methods out of the existing ancestors into new ones. Simple tasks like test methods can be handled by making a copy, dragging the method into the copy, then changing it. Unlike traditional systems, only the changed object has the new code, and nothing has to be rebuilt in order to test it. If the method works, it can simply be dragged back into the ancestor.

Performance

Self VMs achieved performance of approximately half the speed of optimised C on some benchmarks.[6]

This was achieved by just-in-time compilation techniques which were pioneered and improved in Self research to make a high level language perform this well.

Garbage collection

The garbage collector for Self uses generational garbage collection which segregates objects by age. By using the memory management system to record page writes a write-barrier can be maintained. This technique gives excellent performance, although after running for some time a full garbage collection can occur, taking considerable time.

Optimizations

The run time system selectively flattens call structures. This gives modest speedups in itself, but allows extensive caching of type information and multiple versions of code for different caller types. This removes the need to do many method lookups and permits conditional branch statements and hard-coded calls to be inserted- often giving C-like performance with no loss of generality at the language level, but on a fully garbage collected system.[7]

See also

- Cecil (programming language)

References

1. Ungar, David; Smith, Randall B. (2007). "Self". *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages (HOPL III)*. doi:10.1145/1238844.1238853 (https://doi.org/10.1145%2F1238844.1238853). ISBN 9781595937667. S2CID 220937663 (https://api.semanticscholar.org/CorpusID:220937663).
2. "Self "Mandarin" 2017.1" (https://web.archive.org/web/20170524053153/https://blog.selflanguage.org/2017/05/24/self-mandarin-2017-1/). 24 May 2017. Archived from the original (https://blog.selflanguage.org/2017/05/24/self-mandarin-2017-1/) on 24 May 2017. Retrieved 24 May 2017.
3. Wolczko, Mario (1996). "self includes: Smalltalk". *Workshop on Prototype-Based Languages, ECOOP '96, Linz, Austria*.
4. "Self 4.4 released" (https://web.archive.org/web/20171205194557/https://blog.selflanguage.org/2010/07/16/self-4-4-released/). 16 July 2010. Archived from the original (https://blog.selflanguage.org/2010/07/16/self-4-4-released/) on 5 December 2017. Retrieved 24 May 2017.
5. "Self Mallard (4.5.0) released" (https://web.archive.org/web/20171206074534/https://blog.selflanguage.org/2014/01/12/self-mallard-4-5-0-released/). 12 January 2014. Archived from the original (http://blog.selflanguage.org/2014/01/12/self-mallard-4-5-0-released/) on 6 December 2017. Retrieved 24 May 2017.
6. Agesen, Ole (March 1997). "Design and Implementation of Pep, a Java Just-In-Time Translator" (https://web.archive.org/web/20061124224739/http://research.sun.com/jtech/pubs/97-pep.ps). *sun.com*. Archived from the original (http://research.sun.com/jtech/pubs/97-pep.ps) on November 24, 2006.
7. [1] (http://www.sunlabs.com/research/self/papers/chambers-thesis/thesis.ps.Z)

Further reading

- Published papers on Self (http://bibliography.selflanguage.org/)
- Chambers, C. (1992), *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, Stanford University, CiteSeerX 10.1.1.30.1652 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.1652)
- Series of four articles "Environment and the programming language Self" (http://blog.rfox.eu/en/Programming/Series_about_Self.html)

External links

- Official website (http://www.selflanguage.org)
- self (https://github.com/russellallen/self) on GitHub
- Former Self Home Page at Sun Microsystems (https://web.archive.org/web/20020606124955/http://research.sun.com/self/)
- Alternate source of papers on Self from UCSB (mirror for the Sun papers page) (https://web.archive.org/web/20070503053204/http://www.cs.ucsb.edu/~urs/oocsb/self/papers/papers.html)
- Merlin Project (https://web.archive.org/web/20050427111747/http://www.merlintec.com/lsi/)
- Self ported to Linux (without many optimizations) (https://web.archive.org/web/20030613141004/http://gliebe.de/self/index.html)
- Automated Refactoring application on sourceforge.net, written for and in Self (http://selfguru.sourceforge.net/)
- Gordon's Page on Self (http://www.self-support.com/)
- Prometheus object system on the Community Scheme Wiki (http://community.schemewiki.org/?prometheus)
- Video demonstrating self (https://web.archive.org/web/20060813135539/http://www.smalltalk.org.br/movies/)
- dSelf: distributed extension to the delegation and language Self (http://www.ag-nbi.de/research/dself/)
