Elements Report SELF

Kyle Dymowski, Aidan Miller

April 14, 2016

1 The Team

Two people are accredited with the development of SELF, David Ungar and Randy Smith, they began working on the language in 1986 while working at Xerox PARC then in 1987 they moved to Stanford University to build the compiler.

1.1 David Ungar

Dr. Ungar pursued his Ph.D. in Computer Science at the University of California Berkeley from 1980 to 1985. Where he participated in architectural design of SOAR, Smalltalk on a RISC microprocessor. He also modified a simulator and virtual machine to benchmark architectural ideas, and designed, analyzed, implemented and measured the first two generation garbage collector. In 1985 he joined Stanford University as a Assistant Professor where he led the Self team, invented mirror-based reflection architecture and a storage system for prototype-based languages. He then began working at Sun Microsystem Laboratories in 1991 where he continued working on self until 1995. He left sun enterprises in 2006 and started working at IBM where he currently works on the Watson project. [1]

1.2 Randy Smith

Dr. Smith started his education with a double major in physics and mathematics with the highest honor of summa cum laude in 1974 at the University California, Davis. He then pursued his Ph.D. in Theoretical Physics graduating in 1981 from the University of California at San Diego. He began working on self in 1986 at Xerox PARC, then continued his work at Stanford in 1987, and finally finished the project at Sun Microsystems Laboratory in 1991. Where he still works today as a consulting member of the technical staff, he is currently working in the areas of modeling, simulation and optimization. [2]

2 Philosophy

The team that designed Self recognized that object-oriented programming languages were gaining acceptance and becoming widely used, because they offer useful implementation for designing programs. Ungar and team had a different idea on object oriented design leading them to create the first prototype based language. Object oriented languages inherently have uniform access to different kinds of stored and computed data, SELF was looking to extend this and create more expressive power. [3]

In Self, and any other prototype based language, everything is an object, but the difference from languages like C++ is instead of an instance being either a instance of, or a subclass of, it inherits from the relationship by cloning the object. This is how Self eliminates classes, and creates a "inherits from" relationship that describes how objects share behaviour and state. This is done in the cloning step that happens during instantiation, where information formats are copied. This is how SELF creates one of a kind objects that have their own behaviour. Each object is stored in a named slot that can hold state and behaviour. Because every slot is unique it can hold its own states and behaviours, unlike class based systems that have multiple objects with the same behaviour. Classes have no real support for an object to posses it's own behaviours, and there is no way to guarantee that there is only one instance

of the class. Self has neither of these problems, and there is no need to create a new class to extend the behaviour. [3]

Self was designed with not just an object-oriented programming experience, but an object-oriented world view in mind. As programmers are physical beings with concrete sensory experiences, creating a "world of objects illusion" [4] allows program development to occur in a way that is more immediately natural and organic. This inherently encourages expressiveness and conciseness within the language, the end result being an emphasis on simplicity. To demonstrate this philosophy with a more concrete example, Self "does not distinguish state from behavior"[**?**], and accessing a variable is no different than sending a message. This form of dynamic typing, implemented via prototyping, means that cloning objects and modifying their behavior or attributes is an easy way to expand a modeled world - new kinds of objects are easily made from old ones. By reducing the amount of thought a programmer has to put into understanding the specifics of a language, a language becomes more expressive by default. Eliminating the practical differences between a programmer's understanding of what an object is (and how it behaves, etc) and how it is effectively implemented in code is an important step.

3 Innovations

An important aspect of Self is its speed: despite being a prototype based language (which are often inefficient compared with other types of languages), it is able to achieve remarkably fast performance - up to half of that of optimized C. This would not be the case were it not for just-in-time compilation techniques, a methodology which Self was the basis for extensive improvement. Some of these techniques include translating machine code only on demand, caching it, and deleting it when memory is needed, regenerating it as necessary. Another important innovation of Self is generational garbage collection, which acknowledges that the most recently created objects in a program are also those most likely to

fall out of scope soon. In order to make use of this observation, Self separates objects in memory by age, which allows for more immediate disposal of younger objects. While generally fast and effective, this method is not always correct and will not dispose of everything it needs to - "artifacts" may be left behind, unreachable and ultimately useless. Therefore, a different (often slower and more thorough) method must occassionally be implemented in order to keep the program running smoothly. These optimizations, as well as others are what allow Self to perform well, and many of the advancements made in pursuit of performing Self were inherited by languages that would follow - Java adopted JIT compilation and most Java Virtual Machines continue to utilize it. This may ultimately be Self's lasting impact upon the programming landscape, as it is not an extensively well-used language beyond its own community of users.

Having been developed as a dialect of Smalltalk, an older interpretation of object-oriented programming, Self uses a similar virtual machine. That is, programs require their entire memory environment to function, saved as an "image", and can be rather large (and therefore harder to handle) compared to languages like C. The trade-off for this is that Self programs are easier to debug: changes to objects within the system are fully expected and do not require a complete rebuild in order to take effect. This ease of adaptation, combined with Self's prototyping approach to objects, allows the language to achieve its goal of being simple and expressive.

References

[1] David Ungar Software innovator

https://www.linkedin.com/in/davidungar 2016

[2] Randall Smith Consulting Member of Technical Staff

https://labs.oracle.com/pls/apex/f?p=labs:bio:0:118

[3] David Ungar, Randall B. Smith. SELF: The Power of Simplicity. Xerox Palo Alto Research Center, Palo Alto, California 94304

[4] David Ungar Randall Smith

http://bibliography.selflanguage.org/static/programming-as-experience.pdf