将发布在： *LISP AND SYMBOLIC COMPUTATION: An International Journal, 4, 3, 1991*

SELF:简洁的威力*

DAVID UNGAR† (*ungar@self.stanford.edu*)
Computer Systems Laboratory, Stanford University, Stanford, California 94305

RANDALL B. SMITH† (*rsmith@parc.xerox.com*)
Xerox Palo Alto Research Center, Palo Alto, California 94304

翻译：Shining Ray (*shiningray@56.com*)
Nirvana Studio http://www.nirvanastudio.org

摘要：SELF是一个针对探索式编程的面向对象语言，它基于几个很简单和具体的概念：原型、槽和行为。原型结合了继承和实例化，以提供一个比当前大多数面向对象语言更简单更有弹性和框架。槽则将变量和过程统一成一个结构，这就可以令继承层次取代传统语言中的词法范围的功能。最后，因为SELF并不区别对待状态和行为，所以他减少了普通对象、过程和闭包之间的差别。SELF的简洁和表现力给面向对象计算提供了一种全新的视角。

对自己忠诚
——威廉·莎士比亚

To thine own self be true.
—William Shakespeare

1 简介

面向对象的编程语言正在逐渐得到广泛的认可，某种程度上是因为它提供了一种新的很有效的设计计算机程序的观点。然而，他们并不是都提供同一种一致的观点；关于面向对象计算的本质有着很多种不同的看法。在本文中，我们将展示SELF，一种编程语言，它带来了关于对象和消息传递的一种新的视角。就像Smalltalk-80[1]语言 [6]，SELF是为了支持探索式编程 [13] 而设计的，这样包括了运行时类型（也就是没有类型声明）和自动存储回收。

但是与 Smalltalk 不同，SELF 既没有类也没有变量。取而代之的是，SELF 采用了一种原型隐喻来进行对象的创建 [2, 3, 4, 8, 10]。此外，当 Smalltalk 和大多数其它面向对象的语言支持变量访问和消息传送时，SELF 对象通过对“self”对象发送消息来访问他们的状态信息，当前消息的接收器。这很自然地导致了很多发送给“self”的消息，正是由于这些消息，才为这个语言起了这个名字。

面向对象编程的一大能力是在于对不同类型的存储和计算的数据使用了统一的访问手段，SELF 中的概念甚至更加统一化，这就产生了更加强大的表现力。我们相信这些概念将为面向对象计算提供一个更新更有用的视角。

有这样一些原则贯穿了 SELF 的设计：

消息属于底层。SELF把消息传递作为基础操作之一，提供了通过消息单独对存储数据的访问。没有变量，只有包含对象的槽用来返回对象。由于SELF对象只通过发送消息访问状态，所以消息传递对SELF比起其它代变量的语言更加重要更加基础。

奥卡姆的原则（界限简化论）。贯穿在设计中，我们的目标始终集中在概念的精简上：

- 正如上面所说，SELF的设计忽略了类和变量。任何对象都可以扮演一个实例的角色或者是作为共享信息仓库提供服务。
- 访问变量和发送消息没有任何区别。
- 和在Smalltalk中一样，语言的核心没有控制结构。取代的是用闭包和多态来实现任意的控制

*本作品部分是由Xerox所支持，部分由National Science Foundation Presidential Yong Investigator Grant #CCR-8657631, Sun Microsystems, Powell Foundation, IBM, Apple Computer, DEC, NCR, Texas Instruments, 以及 Cray Laboratories 所支持。
†作者目前的住址： Sun Microsystems, 2500 Garcia Avenue, Mountain View, CA 94043.
本论文是 [20] 的一个重要的修订，最初在*OOPSLA '87 Conference Proceedings* (*SIGPLAN Notices*, 22, 12 (1987) 227-241) 上发表。
1Smalltalk-80 是 ParcPlace Systems的商标。在本文中，术语“Smalltalk”将用来指代Smalltalk-80编程语言。

结构。

- 和 Smalltalk中不一样的是，SELF对象和过程是从同一个概念——通过将过程表示为激活记录（activation record）的原型——组织出来的。这个技术使得创建激活记录和创建其它对象的方式一致，通过克隆原型。除了可以共享同一种创建模型之外，过程也可以和普通对象一样存储它们的变量、维护它们的环境信息。在第4节会讲到。

具体化. 我们的经验让我们总结出一个隐喻，元素尽可能具体的 [14, 15]。所以，在类对比原型的问题上面，我们选择了尝试原型。这产生了一个基本的不同点——新对象的创建方式。在基于类的语言中，一个对象是通过在他的类中实例化一个方案。而在像SELF这种基于原型的语言，一个对象可以通过克隆（复制）一个原型来创建。在SELF中，任何对象都可以被克隆。

	基于类的系统	*SELF：没有类*
继承关系	是...的实例 是...的子类	从...继承
创建的隐喻	根据方案构建	克隆一个对象
初始化	执行一个“计划”	复制一个案例
种类关系	需要一个额外的类的对象	不需要额外的对象
无限“元”递归	类的类的类....	不需要

这份论文剩下的部分将详细描述SELF，并且通过两个例子进行总结。我们使用Smalltalk作为我们的参考物，因为他是目前最为广泛了解的语言中一切都是对象的语言。这样，熟悉Smalltalk会对读者更加有帮助。

2 原型：将类和实例混合起来

与C++、Simula、Loops和Ada不同，在Smalltalk中，一切都是对象，每一个对象都包含一个指向他的类的指针，描述了他的格式和保存它的行为的一个对象（见图1）。同样，在SELF中，任何事物也都是对象。但是，SELF对象包含了命名槽而不是类指针，这个命名槽可以保存状态或者是行为。如果一个对象接收到一个消息同时它又没有匹配的槽，这时将通过一个 *parent* 指针继续搜索相应的槽。这就是SELF如何实现继承的。SELF中的继承让对象之间可以共享行为，这样反过来可以让程序员一个简单的更改就可以改变很多对象的行为。例如，一个point[2]对象可能有两个槽来存储他的独特的性质：x和y（见图1）。他的父对象可能是一个对象可能包含了在所有point之间共享的行为：+、-等等。

2 本论文中我们都会举点对象作为例子。Smalltalk中，一个点（point）代表二维笛卡儿坐标系中的一个点。它包含两个实例变量：x坐标和y坐标

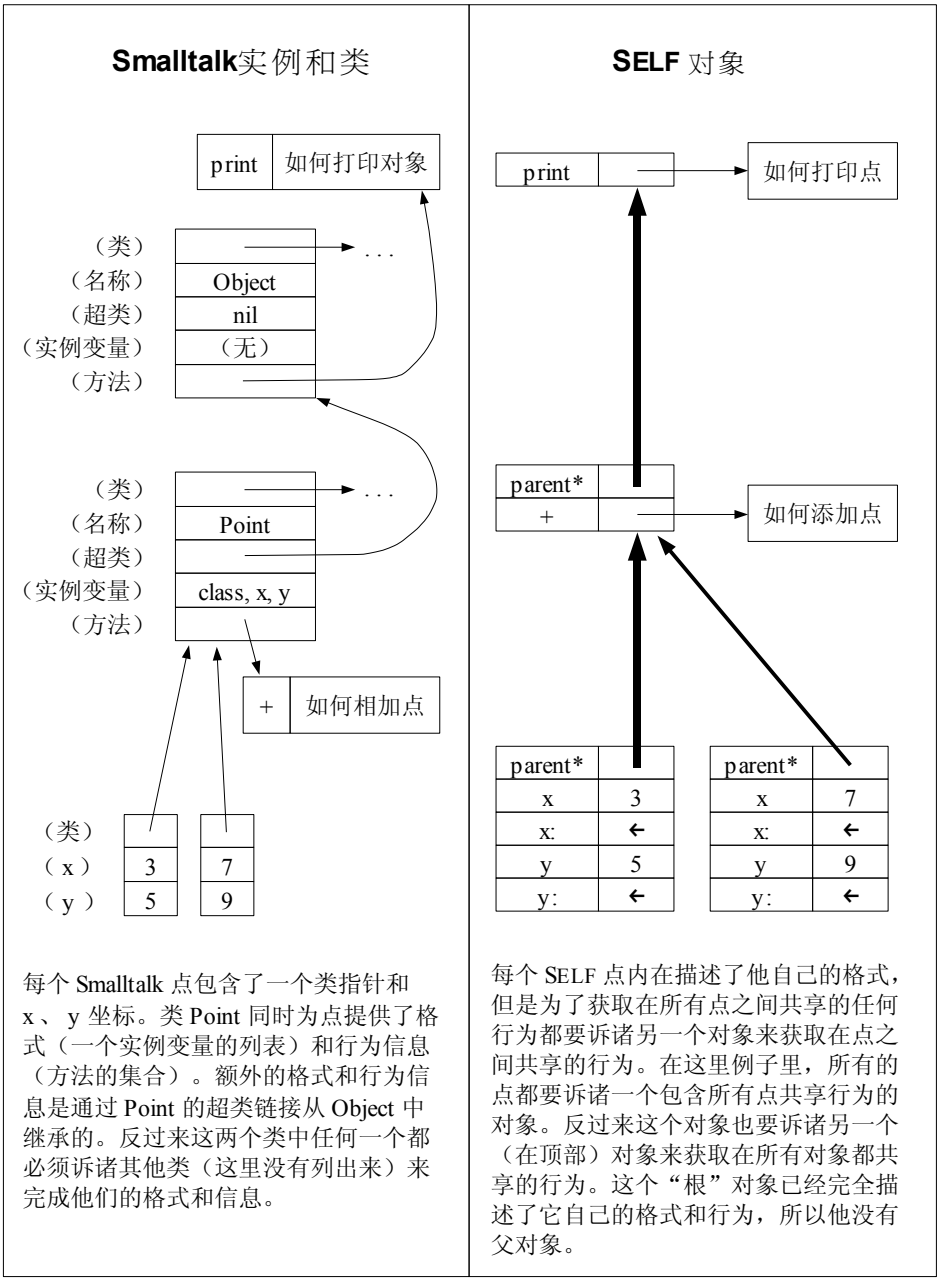

图 1. Smalltalk 的实例、类和 SELF 对象之间的对比
在两边图的底部的是由用户的程序创建的两个点对象

2.1 原型和类的比较

SELF的最有趣的地方之一是他将继承、原型和对象创建结合起来的方式，从而消除了对类的需要（见上面的表格）。

更简单的关系. 原型可以简化对象之间的关系。想象对象在基于类的语言中的行为方式，任何一个都要牵扯到两个关系：“是一个”关系，它指出了一个对象是某一个类的实例，以及“是一种”关系，它指出了一个对象的类是某个其它对象的类的子类。在SELF这种用原型取代类的系统中，只存在一个关系——“从……继承”，他描述了对象之间如何共享行为和状态。这种结构上的简化使得语言更容易理解同时更容易系统化继承层次。

一个运行中的系统可以让我们发现类一样的对象是否足够有用，可以在没有语言的激励下，就让程序员创建他们。没有类－实例之间的区别也许让他更难于理解对象如何单独存在并为其它对象提供

共享信息，也许SELF程序员会创建一种全新组织起来的结构。无论何种情况，SELF的弹性对编程环境提出了一个挑战；他要包含一些进行浏览和描述性的辅助功能。[3]

通过复制来创建. 从原型创建新的对象是通过一个简单的操作——复制——来完成的，我们用一个生物学上的词来比喻它——克隆。从类中创建新的对象是通过实例化来完成的，它包括了对类中的格式信息的描述（见图2）。实例化和按照设计图造一幢房子很相似。对我们来说，复制要比实例化更加简单。

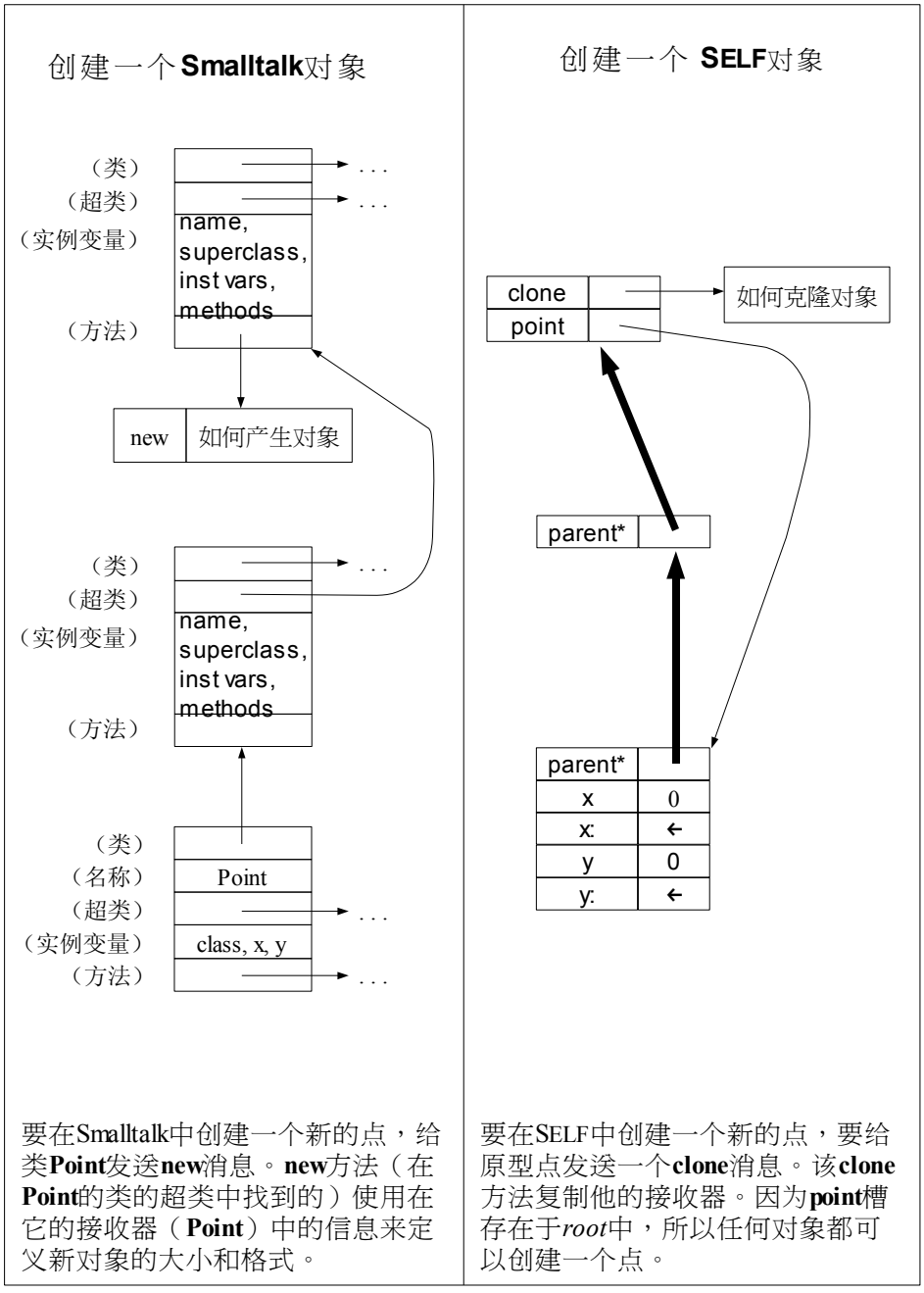

图 2. Smalltalk 和 SELF 中的对象创建

预先存在的模块. 原型比类更加具体，因为它们是对象的*例子*而不是格式和初始状态的*描述*。这些例子会帮助用户复用模块，因为他们更容易理解。一个基于原型的系统可以让用户去调查一个典型代表而非要求他去完全搞清它的描述。

支持“一种只有一个”的对象。 SELF 提供了一个可以方便地包含“一中只有一个”的对象。由于每个对象都有命名槽，并且槽可以保存状态或行为，任何对象都可以有唯一的槽或行为（见图3）。基于

3 从这个第一次写的时候，这个结构已经演化，关于他的叙述参见 [19]

类的系统是为有很多具有相同行为的情况而设计的。没有从语言上对对象独占它自己唯一行为的支持，同时也很难创建只有一个唯一实例的类。SELF 不会有这样两种缺陷。任何对象可以对它自己的行为进行自定义。唯一的对象可以保存唯一的行为，一个单独的“实例”是没有必要的。

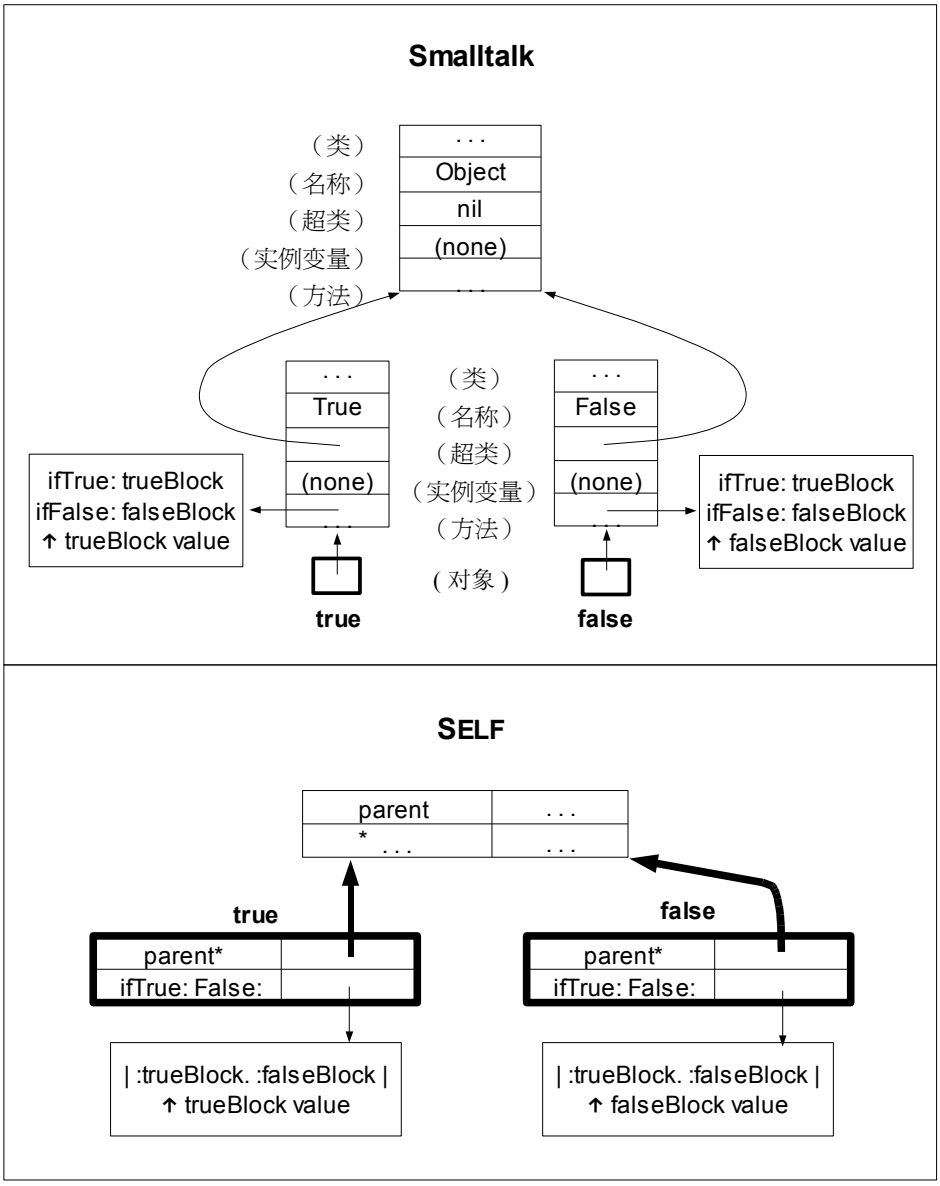

图 3. 在 SELF 中，比起像 Smalltalk 这样的基于类的系统来说，更容易定义唯一的对象。考虑一下表示 true 和 false 布尔值的对象。对于这两个对象，一个系统对其中任何一个都只需要一个实例，但是在 Smalltalk 中，对于每个实例，都要有一个类。在 SELF 中，由于任何对象都可以包含行为，可以直接创建专门针对 true 和 false 的对象。

消除“元”递归。在一个基于类的系统中，没有一个类可以是自给自足的；一定需要另一个对象（它的类）来表达它的结构和行为。这导致了概念上的无限“元”递归：一个point是类Point的一个实例，Point又是一个元类Point的实例，元类Point又是一个元元类Point的实例，然后无限下去[注]。另一方面，在基于原型的系统中，对象可以包含它自身的行为，无需其它对象使它存活。原型消除了“元”递归。

本文中关于原型的讨论正如SELF中所实现的那样自然地应用在它们身上。而没有继承的基于原型的系统可能会有一个问题：每个对象都必须包含所有它自身的行为——就像真实的世界一样——同时这样的系统会放弃计算机和真实世界之间的一个最好的区别——通过改变共享的行为来产生全面的

注 这个例子没说清楚，应该是对象 point 是类 Point 的实例，类 Point 是元类 Class 的实例，元类 Class 是元元类 MetaClass 的实例——译者

更改。
一旦将继承引入了这个语言中，一种很自然的倾向就是将原型做为同一个包含那种对象行为的对象。例如，所有的点的行为可以通过改变原型点的行为来改变。不幸的是，这样一个系统必须提供两种创建对象的方法：一个用来通过原型产生对象，另一个用于复制非原型的对象。最终的结果是导致原型变得十分特殊且一点也不像“样板”。
我们的解决方法是把一个对象家族共享的行为放入一个单独的对象中，将这个对象做为所有那些对象的父对象，包括原型。这样这个原型和所有其它这个家族的成员都是绝对相同的。而包含了共享行为的对象则扮演了类似于类的角色，除了他不包含任何表现形式的信息；它仅仅是保存了一些共享的信息。这样要给SELF中所有的点添加某种行为，我们可以把这个行为添加到这个所有点的父对象中。

3 混合状态和行为

在SELF中，没有直接的途径去访问一个变量；取代的是，靠向对象发送消息来进行访问。所以，要访问一个点对象的“x”值，要向他自己发送“x”消息。这个消息找到“x”槽，并且对在其中发现的对象进行计算。由于这个槽包含一个数字，所以计算的结果就是这个数字本身。若要更改槽“x”的内容，比如改成17，我们不是进行一个像“x←17”之类的赋值操作，而是必须要point对象给自己发送一个“x:”消息，并用17作为参数。point对象必须包含一个叫“x:”的槽，它包含了一个赋值的原语。当然，所有这些发送给“self”的消息将使程序变得很冗长，所以我们的语法允许省略“self”。这样，在SELF中通过消息访问状态变得和Smalltalk中直接访问变量一样简单；“x”可以访问同名的槽，“x: 17”会把17存入槽中。
通过消息访问状态使得继承更加强大。假设我们要创建一种新的点，它的“x”坐标是一个随机数而不是一个预先存好的值。我们复制原来的标准点对象point，删除“x:”槽（这样“x”就不能被更改了），同时用生成随机数的代码替换“x”槽中的内容（见图4）。如果不修改“x”槽而用“halt”方法替换“x:”槽，我们会在运行时得到一个断点。这样，SELF可以表达和活动变量和后台相关的习惯用语。通过消息访问状态也让它更容易分享状态。要创建两个共享同一个“x”坐标的两个点point，可以把“x”和“x:”槽放入一个单独的对象中，并作为这两个点的父对象（见图4）。

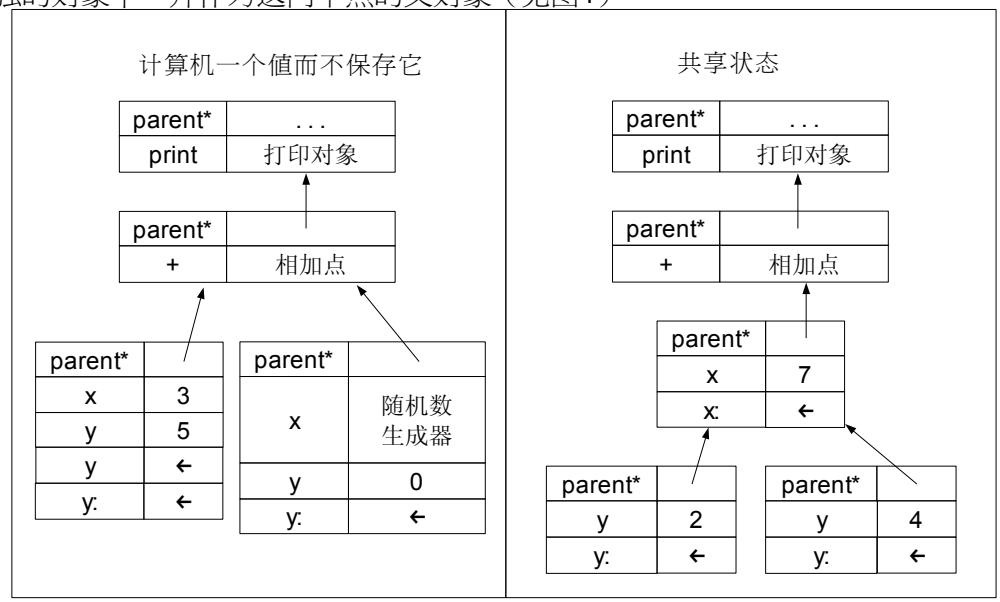

图 4 。两种 SELF 中弹性的例子。左边的是一个 x 坐标随机生成的点。由于所有和 point 相关的代码都是发送消息来获得 x 的值，随机的 x 点仍然可以复用所有已经存在的点代码。右边是两个点共享同一个 x 变量。

在大多数面向对象语言中，访问变量相对于发送消息是一个不同的操作。赋值和读取减弱了消息传送的计算模型。结果就是，消息传递就变得没那么强大了。例如，变量的引入让特化（扩展子类）对象用一个计算结果替换一个变量变得困难。同时，基于类的语言一般把实例变量的名称和顺序存

放在对象的类中（如图1所示）。这更进一步限制了继承的能力；一个类中的规格画蛇添足地约束了实例的格式。最后，变量访问需要范围规则，也更加复杂化。例如，Smalltalk有五种变量：局部变量（临时变量），实例变量、类变量、池变量和全局变量，他们的范围大致对应于实例化的等级层次。

4 闭包和方法

Lisp社群已经用闭包（也就是Lambda算式）作为控制结构的基础 [1, 17] 取得了杰出的成就。Smalltalk的块的经验也验证了这一点；闭包提供了一个强大，而又易于使用的隐喻，让用户可以发掘并定义他们自己的控制结构。甚至，这个能力对于任何支持用户自定义抽象数据类型的语言，都是至关重要的。在SELF中，闭包是由包含一个环境链接和“value”、“value:”、“value:With:”等等如此类的方法（由参数的数量决定）的对象来表示的。

局部变量. 方法需要存储局部变量，在SELF中，方法的槽可以实现这个功能。在Smalltalk中，一个方法会创建一个激活记录，它的初始内容是由方法所描述的。例如，在方法中列出的临时变量的数量描述了应该在激活记录中保留的字段的数量，以便保存变量。这与类包含用于实例化他的实例对象的结构化的描述的方式是很相似的。但在SELF中，扮演方法的角色的对象是激活记录的原型；通过复制和调用它们来运行子过程。所以，局部变量是通过在激活记录中为他们保留槽来分配的。这样一个好处是原型的槽可以初始化为任何值——他们甚至可以包含私有方法和闭包（块）。

环境链接. 一般来说，一个方法必须包含一个指向他所包围的闭包或范围的链接。这个链接是用来解决没有在方法本身里面找到的变量的引用。SELF不使用分离的范围信息，而是用一个方法中的 *parent* 链接来实现这个功能。如果在当前的范围中没有找到某个槽，那么就会跟随parent链接继续查询下一个更外面的范围。这里有一些有趣的又是必要的机制来处理词法范围。首先，parent链接要被设置到合适的对象上。这对于一个普通对象来说很简单，作为克隆的结果之一，parent链接会被自动设为它的原型的parent。对于方法，由编译器所创建的对象是用作一个原型激活，且当他被调用时，就被克隆。克隆出来的对象的parent会被设置为消息的接收器。在这样方式中，方法的范围是嵌在接收器的范围中的。对于SELF块，当创建块时，一个环境链接会设置到这个块所包围的激活记录上。然后，当这个块被激活时，它的方法的parent链接就会设为这个块的环境链接。

第二，为了能让槽包含局部变量，而且就像访问其他东西一样访问这些变量，隐含的“self”操作数必须具有一个特殊的含义：从当前的激活记录开始进行消息查找，但要把消息的接收器设置为当前的接收器。在某种程度上，这种方式和Smalltalk中的“super”概念是相反的，“super”是从接收器的超类开始进行消息查找（见图5）。

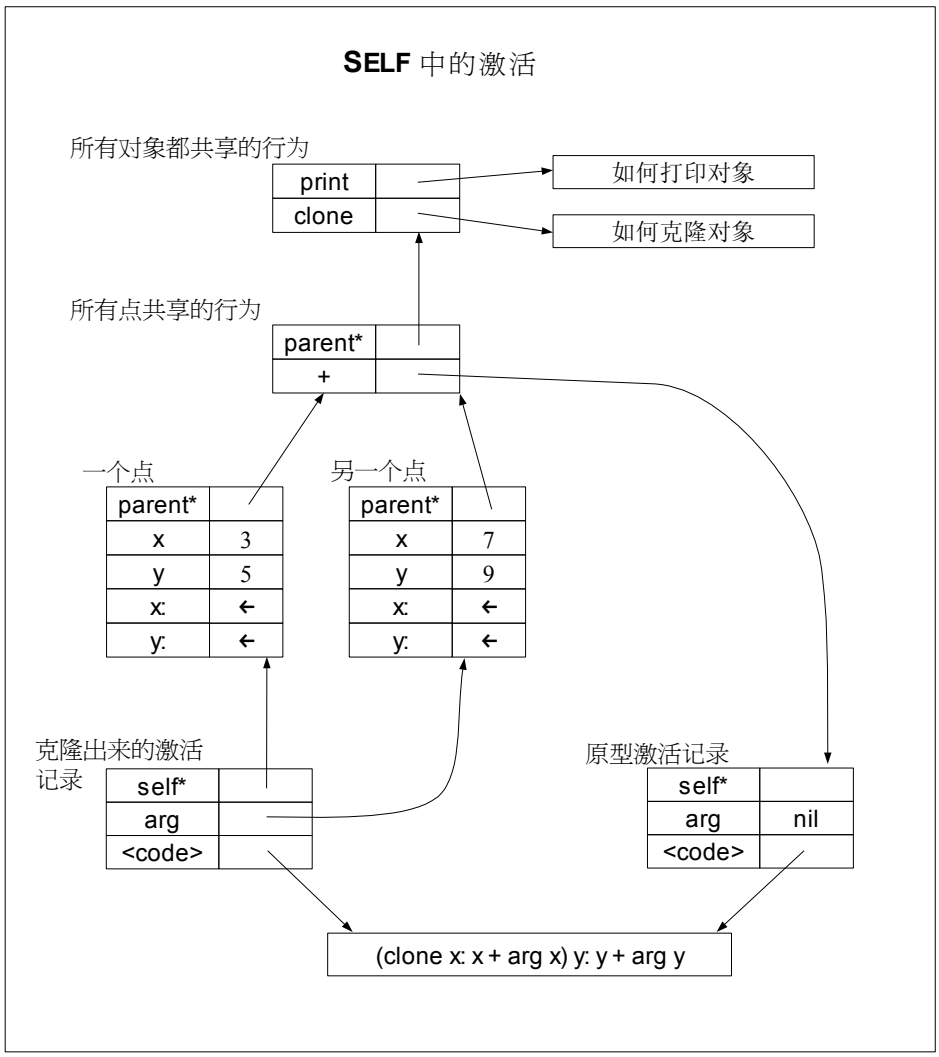

图 5. 这幅图展示了当向点 (3, 5) 发送带 (7, 9) 参数的“ +” 消息时发生了什么。加法的查询首先从 (3, 5) 开始并且在对象中发现了一个匹配的槽。由于这个槽的内容是一个方法对象，所以他就被克隆，克隆后的对象的参数槽便设置为消息的参数，同时他的 parent 设在了接收器上。当“ +” 的代码执行时，对 x 的查找将跟随从当前激活记录开始的继承链最后发现接收器的 x 槽。他同时也通过同样的方法发现了“ arg” 槽的内容。正是因为有了这个从当前激活开始查询隐含的“ self” 接收器的技术， SELF 中统一了局部变量、实例变量和方法查询。

5 推论：SELF 将去何方？

在设计SELF时，我们常常发现一些很奇怪的不断重现的主题。我们把它们展示给读者，以便大家思考。

行为主义. 在很多对象语言中，对象是被动的；一个对象表示“它是什么”。在SELF中，一个对象表示“它做什么”。由于变量访问和消息传递一样，普通的被动对象可以看作只是返回自身的方法。例如，我们看看数字17。在Smalltalk中，数字17代表了一个特定的（不可变的）状态。在SELF中，数字17仅仅是一个返回自身的对象，同时按照某种和算术有关的方式进行运转。唯一的了解一个对象方法是去看他的行为。

把计算看作是细化. 在 Smalltalk中，数字17是一个有着特定状态的数字，而且状态信息是用于算术原语的——如加法。在SELF中，可以把17看作是对数字的共享行为的细化，它通过返回比参数大17的结果响应加法来响应加法。由于在SELF中，一个激活记录的父对象是设置到消息的接收器上的，所以方法激活可以看做是看作是对于一个接收器的短暂*细化*的创建。同样的，块（闭包）激活可以看作是对所包围的上下文范围的激活记录的一个*细化*的创建。

在我们的例子里，我们把点的共享行为对象作为一个普通的*被动*对象。另一种手法是从*方法*中建立出类“类”对象。在SELF中，这个点的共享行为对象可以是仅仅一个方法，用来返回一个样板点的克隆点。然后我们可以把这个方法安装在根对象的“point”槽中。这样一个对象可以担任两种角色：他的代码可以创建新的点，而且他的槽（局部的）可以用来存放点的共享行为。对于这种写法，我们不认为是构建一个系统的最佳途径，但是使用方法来存放一组对象的共享行为也是SELF可以给予的弹性的例子。

将父对象看作是共享部件。最后，可以把一个对象的祖先对象看作是一个这个对象的共享部件。从这个观点出发，一个SELF点包含一个私有的部分，里面是 x 和 y 槽，一个和其他点对象共享的部分，包含了 +、-等等其他槽，还有一个和其他所有对象一起共享的部分，包含了所有对象共同的行为。将祖先对象看作是共享部件的观点拓宽了继承的适用性。

6 语法

在这一节中我们会描绘SELF对象文本表现的语法的大体轮廓。可能的地方，我们会附上Smalltalk的语法来避免混淆。我们加入了一个槽列表语法以便可以直接在代码中创建对象。一般来说，SELF对象是写在圆括号中的，并包括一个槽的列表，如果是方法，还包括代码。块和方法一样，但是写在方括号中间而不是圆括号。

代码部分基本按照Smalltalk的语法，但要注意一些区别：当接收器是“self”的时候，一般忽略。方法的返回值总是最后一个表达式的结果。关键词消息是从右到左结合。使用了新的大小写规则为了让发送关键词消息更易阅读：第一个关键词首字母必须小写，同一个选择器中的后续关键词首字母要大写。这些改变减少了SELF代码中的括号的数量和其它词法上的干扰。槽列表语法在Smalltalk中有一个小小的祖先。如果出现了槽列表，必须把他们放入一对竖条中。而且槽列表中的每个条目必须用句号分开（最后一个句号是可省的）。最后，有几种槽的形式：

- 一个只有名字的选择器表示有两个槽：一个槽初始化为*nil*，另一个槽的名字后面再加上一个冒号，被初始化为赋值原语（由←表示）。
  例如，这个对象
  (| x. |)
  包含两个槽：一个叫作 **x** ，包含空，另一个叫做 **x:** 包含 ← 。这和声明一个Smalltalk变量的效果一样。
- 一个后面跟着左箭头（“<-”）以及一个表达式的选择器也表示两个槽：一个初始化为这个表达式的值的槽，以及相应的赋值槽。他的效果类似于一个初始化的变量。例如，这个方法对象

```
(
    | tally <- 0 |
    10 do: [tally: tally + random].
    tally
)
```

  （这里“random”是一个根对象中的槽，返回一个随机数）返回 10 个随机数的和。它包含了一个叫做“tally”的槽，被初始化为 0，以及一个叫做“tally:”的槽包含了赋值原语。
- 后面跟着等于号（“=”）和一个表达式的选择器只表示一个槽，它会被初始化为表示式的值。它和左箭头形式的效果在其中一方面是一样的，除了变量是只读的（不会创建赋值槽）。
- 最后，一个前面有冒号的一元选择器（也就是标示符）定义了绑定到相应消息参数上的槽。例如，“[| :a. :b | a < b]”定义了一个有两个参数“a”和“b”的块。
  方法的参数也可以移到选择器中，就像Smalltalk中一样：

```
display:At: = (
    | :aForm. :aPoint |
    (bitblt copy
        destination: self At: aPoint Source: aForm)
    copybits)
```

  和

```
display: aForm At: aPoint = (
    (bitblt copy
        destination: self At: aPoint Source: aForm)
    copybits)
```

是等价的。

7 例子

第一个例子十分简单。它假设系统是分成两类对象的：存放共享特性的对象（traits object），和原型。（在例子中，原语 _AddSlotsIfAbsent: 会把槽加入它的接收器中当槽不存在的时候；原语 _Define: 在原来的位置上重新定义它的接收器）

```
_AddSlotsIfAbsent: ( |
    traits = ().                        创建一个对象来存放特性对象。
    prototypes* = ().                   创建一个对象来存放原型对象。它是一个父对象以便更容
| )                                       易引用它的内容。
traits _AddSlotsIfAbsent: ( | clonable = () | )
traits clonable _Define: ( |            定义存放可克隆对象特性的对象。
    copy = ( _Clone ).                  定义一个调用“clone”原语的复制方法。
| )
traits _AddSlotsIfAbsent: ( | point = () | )
traits point _Define: ( |               定义存放点（point）的特性的对象。
    parent* = traits clonable.          继承复制方法
    printString = (                     定义一个方法来构造一个可打印的字符串。
        x printString, '@',             连接代表 x 和 y 的字符串，
            y printString ).            并用“@”分隔。
    + aPoint = (                        定义加法的方法。
        | newPoint |                    使用一个局部槽（newPoint）。
        newPoint: copy.                 复制接收器来作为返回结果.。
        newPoint x: x + aPoint x.       设置它的x坐标。
        newPoint y: y + aPoint y.       设置它的y坐标。
        newPoint ).                     返回新的点。
    - aPoint = (                        利用赋值（如 x:, y:）将返回自身的性质来定义减法。
        (copy x: x - aPoint x)          （为了方便，方法只要可能就应该返回自身self）
            y: y - aPoint y ).
| )
prototypes _AddSlotsIfAbsent: ( | point = () | )
point _Define: ( |                      定义原型点。
    parent* = traits point.             通过只读的parent槽来继承共享特性。
    x <- 0.                             定义读/写槽 x 和 y 并初始化为0。（也就是，带着赋值原
    y <- 0.                               语同时定义 x: 和 y:）。
| )
traits integer _AddSlots: ( |           添加一个行为，可以从已存在的整数特性对象中创建点
    @ y = ( (point copy x: self) y: y
).
| )
```

下面的例子更加有趣；它演示了如何充分利用SELF的独特功能来建立一个存放二叉树对象的数据结构。

```
traits _AddSlotsIfAbsent: ( | emptyTree = () | )
traits emptyTree _Define: ( |               定义一个保存空树特性的对象。
    parent* = traits clonable.              继承复制方法
    includes: x = ( false ).                空树不会包含任何东西
    insert: x = (                           创建一个新的树节点treeNode，并设置它的内
        parent: treeNode copy                 容，
        contents: x ).                      并更换自己的父对象。使用了动态继承。
    size = 0.
    do: aBlock = ( self ).
| )
prototypes _AddSlotsIfAbsent: ( | tree = () | )
```

```
tree _Define: ( |
    parent* <- traits emptyTree.
| )
```

定义一个空的树作为原型。它有一个可赋值的 *parent* 槽，已经设置为 *traits emptyTree*。

```
traits _AddSlotsIfAbsent: ( | treeNode = () | )
```

```
traits treeNode _Define: ( |
    parent* = traits clonable.
    includes: x = ( | subT |
        x = contents ifTrue: [^true].


        subT: x < contents
            ifTrue: [left] False: [right].
        subT includes: x ).
    insert: x = ( | subT |
        x = contents ifTrue: [^self].
        subT: x < contents
            ifTrue: [left] False: [right].
        subT insert: x.
        self ).
    size = ( left size + 1 + right size ).
    do: aBlock = (
        left do: aBlock.
        aBlock value: contents.
        right do: aBlock.
        self ).
    copy = (
        (resend.copy left: left copy)
            right: right copy ).
| )
```

定义保存树节点特性的对象。
继承复制方法。
这个方法使用了一个局部变量 *subT*。
给自身发送内容并进行比较。使用*SELF*的功能
继承状态（内容槽）。
如果找到了返回真。
选择要递归的子树。

递归并返回递归调用的结果。
把 *x* 插入接收器中；和*includes:*一样递归。

当复制一个树节点的时候同时复制整个子树
当一个树节点是为了插入树而复制的时候，这
提供了一个新的空树

```
prototypes _AddSlotsIfAbsent: ( | treeNode = () | )
```

```
treeNode _Define: ( |
    parent* = traits treeNode.
    left <- tree.
    right <- tree.
    contents.
| )
```

定义样板树节点。
一个不变的*parent*槽。
可以赋值的子树槽，初始化为原型（空）树。

可赋值的内容槽。

8 相关的工作

我们要对 Xerox PARC 的 System Concepts Laboratory 的过去和现在的成员表示我们深深的感谢，是他们用 Smalltalk 点亮了这条前进的道路 [6]。SELF 通过消息传送来访问变量的方式，很大程度上应归功于和 Peter Deutsch 之间的交流，其中他谈及了过去早些时候他称之为“O”的一种未公布的语言。一些 Smalltalk 程序员也采用了这种形式的变量访问 [11]。Trellis/Owl 是一种独立设计的面向对象语言，它包含了让消息调用看上去像元素访问和赋值的语法糖（好的语法形式）[12]。这和我们的方式刚好相反。我们仍然坚持在 SELF 中使用消息传送的语法是为了强调行为的内涵。Strobe 是一个为 AI 设计的基于框架的语言，它同样把数据和行为混合在了槽中。Loops，一种 InterLisp 的带对象的扩展，也包含了活动变量 [18]。
我们还要感谢 Henry Lieberman 引起我们对于原型的关注 [10]。不像SELF， Lieberman 的原型同时包含了共享信息。他的克隆对象继承它们的原型，同时即时添加私有槽。尽管这个方法避免了对特性对象的需要，但它的原型相对于SELF中的是更加重量级的对象。在一个给Smalltalk添加基于原型的对象层次的项目中，把原型称之为范例（exemplar）。就像我们给SELF的设计一样，对象是通过克隆范例来创建的，同时同一个数据类型的多重表现形式也是允许的。但和SELF不一样的是，这个系统仍然包含了类作为一个抽象类型层次，以及多继承的两种形式。一个有趣的贡献是这个范例系统支持一种“或”继承。而SELF看上去比范例系统更加反传统，有两个方面：它从语言中消除了变量访

问，同时它统一了对象、类和方法。
Alternate Reality Kit [14, 15] 是一个直接基于原型和活动对象的操作模拟环境，同时它给我们很多洞察原型世界的能力。当我们还在努力去想领会原型的内在时 ，Alan Borning对于基于原型的环境的经验，尤其是ThingLab [2, 3, 4]让他成为一个极好的宣传员。
DeltaTalk草案 [5] 包括了融合Smalltalk方法和块的一些想法，它帮助我们了解了在这方面存在的问题。Actors [7] 系统包含了活动对象，但它们是进程，不像SELF的过程模型。Actors也抛弃了类，使用代理替换了继承。Oaklisp [9] 是一个带有底层消息传递的Scheme版本。然而，Oaklisp 是基于类的而且继承层次是和词法嵌套相独立的；还有他似乎没有融合lambda表达式和对象。

9 总结

SELF 提供了面向对象语言的一个新的典范，它将简洁和表达力很好得结合在了一起。他的简洁源于认识到类和变量是不必要的。这两个东西的消除驱逐了元类递归，解除了在实例化和新建子类之间容易让人出错的区别，同时模糊了对象、过程和闭包之间的界限。减少一个语言中的基本概念可以让这个语言更加容易解释、理解和使用。然而，使语言更简单和使系统组织清晰之间仍然有个对立的平衡关系。当模型的多样性减少时，语言方面对系统结构上的指引线索的多样性也同样减少了。 。

让 SELF 更加简单的同时也使得他更加强大。SELF 可以表达从传统的面向对象语言中来的一些习惯用语，如类和实例，甚至可以超过这些，比如表达“一种只有一个”对象，活动值（active value），内联对象和类，以及覆盖实例变量。我们相信 SELF 中提出的一些计划和想法，提供了洞察面向对象计算本质的能力。

10 鸣谢

我们要感谢Daniel Weise和Mark Miller耐心的聆听以及对我们在Scheme上的指导。Craig Chambers、Martin Rinard和Elgin Lee也参与了对语言的浓缩和提炼。最后，我们还要感谢所有的读者和评论员，感谢你们提出了很多很有帮助的建议和批评，特别是Dave Robson，他帮助了进行去粗取精。

参考资料

1. Abelson, H., Sussman, G. J., and Sussman, J. *Structure and Interpretation of Computer Programs,* MIT Press (1984).
2. Borning, A. H. *ThingLab—A Constraint-Oriented Simulation Laboratory.* Ph.D. dissertation, Stanford University (1979).
3. Borning, A. H. The Programming Language Aspects of ThingLab, A Constraint-Oriented Simulation Laboratory. In *ACM Transactions on Programming Languages and Systems*, 3, 4 (1981) 353-387.
4. Borning, A. H. Classes Versus Prototypes in Object-Oriented Languages. In *Proceedings of the ACM/IEEE Fall Joint Computer Conference* (1986) 36-40.
5. Borning, A., and O’Shea, T. DeltaTalk: An Empirically and Aesthetically Motivated Simplification of the Smalltalk-80™ Language. Unpublished manuscript (1986).
6. Goldberg, A., and Robson, D. *Smalltalk-80: The Language and Its Implementation*. Addison-Wesley, Reading, MA (1983).
7. Hewitt, C., and Agha, G. ACTORS: A Conceptual Foundation For Concurrent Object-Oriented Programming. Unpublished draft, MIT Artificial Intelligence Laboratory (1987).
8. LaLonde, W. R., Thomas, D. A., and Pugh, J. R. An Exemplar Based Smalltalk. In *OOPSLA ’86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 322-330.
9. Lang, K. J., and Pearlmutter, B. A. Oaklisp: An Object-Oriented Scheme with First Class Types. In *OOPSLA ’86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 30-37.

10. Lieberman, H. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 214-223.
11. Rochat, R. In Search of Good Smalltalk Programming Style. Technical Report No. CR-86-19, Computer Research Laboratory, Tektronix Laboratories, Beaverton, OR (1986).
12. Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An Introduction to Trellis/Owl. In *OOPSLA '86 Conference Proceedings*. Published as *SIGPLAN Notices*, 21, 11 (1986) 9-16.
13. Sheil, B. Power Tools for Programmers. *Datamation*, 29, 2 (1983) 131-144.
14. Smith, R. B. The Alternate Reality Kit: An Animated Environment for Creating Interactive Simulations. In *Proceedings of 1986 IEEE Computer Society Workshop on Visual Languages* (1986) 99-106.
15. Smith, R. B. Experiences with the Alternate Reality Kit: An Example of the Tension Between Literalism and Magic. In *Proceedings of the CHI+GI '87 Conference* (1987) 61-67.
16. Smith, R. G. Strobe: Support for Structured Object Knowledge Representation. In *Proceedings of the 1983 International Joint Conference on Artificial Intelligence* (1983) 855-858.
17. Steele, G. L., Jr. Lambda, the Ultimate Imperative. AI Memo 353, MIT Artificial Intelligence Laboratory (1976).
18. Stefik, M., Bobrow, D., and Kahn, K. Integrating Access-Oriented Programming into a Multiprogramming Environment. *IEEE Software Magazine*, 3, 1 (1986) 10-18.
19. Ungar, D., Chambers, C., Chang, B., and Hölzle, U. Organizing Programs Without Classes. To be published in *Lisp and Symbolic Computation*, 4, 3 (1991).
20. Ungar, D., and Smith, R. B. SELF: The Power of Simplicity. In *OOPSLA '87 Conference Proceedings*. Published as *SIGPLAN Notices*, 22, 12 (1987) 227-241. *20*