A Denotational Semantics of Inheritance

by
William R. Cook
B. S., Tulane University, 1984
Sc. M., Brown University, 1986

Thesis
Submitted in partial fulfillment of the requirements for the
Degree of Doctor of Philosophy in the Department of Computer Science
at Brown University.

This dissertation by William R. Cook is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date 5/20/89 Peter Wegner

Peter Wegner

Recommended to the Graduate Council

Date 5/20/89 Stanley Zdonik (signed in absence by Pete Wegner)

Stan Zdonik

Date 5/18/89 John C Mitchell

John Mitchell
Department of Computer Science
Stanford University

Approved by the Graduate Council

Date 5/20/89 [illegible]

Vita

I was born on November 21, 1963, in Ann Arbor, Michigan. I lived there for two years and have not been back since. As a child I lived in Cambridge, Philadelphia, Chicago, and finally Tampa, after my parents separated. Throughout this migration my best friends were my older brother Victor and my younger brother Christopher. I drew lots of pictures and did what I pleased. I attended the Jesuit High School in Tampa, but left after three years to begin college at Tulane University. That was a good move; I had felt out of place in high-school but found people to love at college. I also came to know my father, who teaches at Tulane. After four years I moved to Providence to attend Brown. The process of graduate school fascinated me. It became clear that merely answering questions or solving known problems well was not enough. Only after being criticized for attacking an interesting problem in the same way as I had attacked problems before did I realize that I should examine how I approached problems. I identified my method up to that time as focusing on the development of "incremental graphical systems" — practical, but there was nothing new to it. I realized that the problem is not the problem: I am the problem. So I started trying to work in new ways. Looking in my journals I can trace the path by which I gradually came upon my thesis, by drawing lots of pictures and looking at what interested me. And then I found that I had a grip on an idea much larger than I was prepared to handle. I discovered that I was poorly educated: unable to write, lacking discipline and thoroughness. For the last year of my writing I moved to California and worked at HP Labs, where I received wonderful support and encouragement. While I have managed to learn to write and discipline myself to some degree, it is a struggle. I am still in awe of the very process of graduate school, but worry that so much stress may not be good for one's mental health.

Preface

This thesis develops a semantic model of inheritance and investigates its applications for the analysis and design of programming languages. Inheritance is a mechanism for incremental programming in the presence of self-reference. This interpretation of inheritance is formalized using traditional techniques of fixed-point theory, resulting in a compositional model of inheritance that is directly applicable to object-oriented languages. Novel applications of inheritance revealed by the model are illustrated to show that inheritance has wider significance beyond object-oriented class inheritance. Constraints induced by self-reference and inheritance are investigated using type theory and yield a formal characterization of abstract classes and a demonstration that the subtype relation is a direct consequence of the basic mechanism of inheritance. The model is proven equivalent to the operational semantics of inheritance embodied by the interpreters of object-oriented languages like Smalltalk. Concise descriptions of inheritance behavior in several object-oriented languages, including Smalltalk, Beta, Simula, and Flavors, are presented in a common framework that facilitates direct comparison of their features.

Acknowledgements

I would like to thank Peter Wegner and my readers, Stan Zdonik and especially John Mitchell, for their constant encouragement and direction. I thank Laura Ide for teaching me how to write technically, and Trina Avery for proofreading. I would also like to thank the group at HP Labs, Alan Snyder, Walt Hill, Walter Olthoff, and Peter Canning, among others, for helping me move on to bigger and better things. My debts to these and other friends, especially my brothers Victor and Chris, Fred Bourgeois, Rachel Buck, Brian Dalio, Page Elmore, Tom Freeman, Craig Hansen-Sturm, Victor Law, Eric Lormand, Leonard Nicholson, James Redfern, Tom Rockwell, Chris Warth, and Felix Yen, cannot be described here. This can be only a public acknowledgement of my deep personal thanks. This work is dedicated to my parents, Linda Hayes and Victor Cook, Jr.

Contents

List of Tables

List of Figures

A Denotational Semantics of Inheritance

William R. Cook

May 18, 1989

Chapter 1

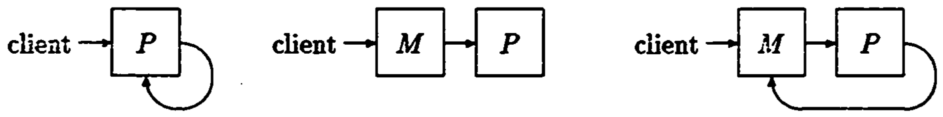
Introduction

This thesis presents a model of inheritance and investigates its applications for the analysis and design of programming languages. The notion of inheritance investigated here was first introduced by Simula, and was later developed in the languages Smalltalk, Flavors, Beta, etc. Although these languages have been in use for over a decade, there has not been a clear consensus on what inheritance is and how it should be modelled.

One reason for this confusion is that there are several different contexts in which inheritance has a different meaning than in object-oriented languages. The most notable example is knowledge representation, where inheritance is form of relationship or inference rule that allows default reasoning. Although formal models are being developed for inheritance in knowledge representation systems, they are not immediately suitable for explaining inheritance in object-oriented programming. This recognition is an important step in separating out the different notions of inheritance that are currently in use. Each notion of inheritance should be given an interpretation that is natural to its own domain: the model of inheritance presented here is firmly based on traditional forms of programming language analysis. Once this is done, comparison of the formal models of different notions of inheritance may then prove fruitful.

1.1 The Inheritance Model

Inheritance is a mechanism for incremental programming in the presence of self-reference. Incremental programming is the definition of new program units by modifying existing ones. A special mechanism is needed for this task because there is a subtle interaction between modification and self-reference, when modification is achieved by derivation of a new structure resembling the original. The problem is that this new structure is a different entity than the original, yet modification only makes sense if the changes and the original are merged together into a single conceptual unit. Inheritance solves this problem, while allowing modification to be distinct from the original, while enforcing a contextual binding of self-reference. This is done by changing self-reference in the original to refer to the modification when the original is referenced by the modification, as illustrated below:



Self-reference *Modification* *Inheritance*

The intuitive explanation of the purpose of inheritance is one of the major contributions of this work.

Collections of named attributes, or records, are developed as a useful framework for describing modification. Records are useful because modifications may be expressed as collections of attributes to be added or replaced. A mechanism is introduced for describing record combination functions that allow different forms of replacement or addition to records.

In order to provide continuity with previous work, the manipulation of self-reference during inheritance is best described within the context of a traditional analysis of self-reference, like fixpoint semantics. The concept of a *generator* is introduced to describe the specifications, or templates, used in fixpoint theory to analyze self-referential definitions. In the traditional use of fixpoint theory only the fixpoints of generators are of interest generators are not used in any other way.

Inheritance is defined as generator derivation, in which self-reference of the derived generator is used as the binding for self-reference in the inherited generators. Several different forms of generator composition are examined, leading to a general mechanism for *wrapper* combination which allows the modification to refer to the result of the combination. The sharing of self-reference is facilitated by the introduction of *distributive operators*, which act as primitive inheritance combinators. Different mechanisms for the modification of record generators are then developed, based on distributive record combination. A framework for multiple inheritance is also built using the model. Generalizations of inheritance that do not require the disciplined sharing of self-reference are proposed.

The model of inheritance severs a number of purposes. Perhaps most surprisingly, it demonstrates that inheritance is an intuitively understandable mechanism that cannot be expressed directly in conventional languages. In addition, it makes formal analysis and evaluation of inheritance possible, using the many well-developed tools associated with fixpoint theory and denotational semantics.

1.2 Applications of Inheritance

The use of the inheritance model is illustrated in a diverse set of applications. The first example is an abstract explanation of the traditional use of inheritance for derivation of classes of objects. This is followed by a novel form of derivation involving recursive constructors of objects (*i.e.* self-referential classes) in which the new constructor is strikingly different from the original. A solution is presented to the long-standing unsolved problem of "memoization", in which a general procedure that memoizes another function is sought. A solution is also sketched to the problem of deriving a modified copy of an entire class hierarchy. Many of the techniques involved in these discussions are currently outside the range of existing systems and languages.

1.3 Type Theory of Inheritance

Type theory is used to analyze the constraints that arise during self-reference and inheritance. The basic constraint is that self-reference in a definition must be compatible with the kind of structure being defined. This constraint is reflected in fixpoint theory, because the fixpoint operator is only defined on generators whose result type (range) is a subtype of its formal parameter type (domain), a condition that guarantees compatibility of self-reference. This constraint provides a direct characterization of 'abstract classes' in object-oriented languages: they represent generators that do not have a fixpoint, and hence do not have a behavior for instances.

Additional constraints arise during inheritance, primarily due to the sharing of self-reference. Since self-reference in the inherited structure is changed to refer to the modifications, the modifications must be compatible with the original self-reference if the inheritor is to be consistent. However, even when type-constraints are satisfied, inheritance may be considered a fundamental violation of encapsulation because of the change of self-reference. A new notion of encapsulation is proposed, which includes a contract on the recursive behavior of a structure.

1.4 Correctness of the Model

Evidence is given for the correctness of the model by showing that it is equivalent to the kind of inheritance used in real object-oriented languages. In these languages, inheritance is defined by a method search or lookup algorithm. To demonstrate that the it is correct, a semantics of message sending based on the denotational model is devised and proven equivalent to the traditional operational semantics based on method lookup. The concept of a method system is introduced to bridge the gap between the minimal context of the generator inheritance model and the fairly special-purpose context of object-oriented languages. A method system may be understood as a snap-shot of the context of an object-oriented program. A simple language for expressing methods is defined, including self-reference and super-reference, but excluding instance variables, assignment, and other aspects not relevant to method lookup. The proof proceeds by demonstrating that the denotational and operational definitions of an 'interpreter' or semantics for method systems are equivalent.

1.5 Inheritance in Denotational Semantics

The model is used to give a denotational semantics for a simple language with classes and inheritance. This language and its analysis serves as a framework in which to define the semantics of inheritance in object-oriented languages in later chapters. The primary novelty in this semantics is the use of an explicit domain of generators, and a loosening of the conventional restrictions on the use of the fixpoint function in standard semantics. These conventions are what prevented previous semantic frameworks from providing a natural semantics for inheritance.

The framework serves as a general context in which inheritance mechanisms in object-oriented languages are analyzed and compared. Each language's characteristic class declaration and instantiation syntax is added to the framework and semantic definitions added to provide a semantics which captures the essential aspects of inheritance in the language, while ignoring irrelevant details. The resulting semantic definitions are easily compared and contrasted. However, in extracting inheritance for study, other features of the languages are given very abstract treatment, or even modified to fit into the common framework. The semantic sketches demonstrate the essential similarity of inheritance in these languages, while clarifying their significant differences.

1.6 Inheritance in Simula

Simula was the first object-oriented language, but it includes a number of features which were dropped from later languages. The semantics of Simula are complicated by its use of *qualification*, which allowed direct access to all ancestors, and by the optional and weak nature of the *virtual* specification. Instances also had an imperative that specified their initial behavior, and could invoke specialized imperatives through the inner command.

1.7 Inheritance in Smalltalk

Smalltalk is in many ways a simplification of Simula, primarily based upon the attribute concept. All methods are considered virtual, and qualification was restricted to allow only relative access to the first ancestor. The imperative was dropped in favor of *metaclasses* for object initialization. Inheritance occurs uniformly at both the class and metaclass level. The semantics of Smalltalk retains this regularity by modelling metaclasses as generators which contain the class generator.

1.8 Inheritance in Beta

Beta was also derived from Simula, but unlike Smalltalk, it uniformly extended the inner construct to attributes, resulting in a degree of 'definitional power' which prevents basic behavior from being redefined. Instead of replacing inherited attributes, a subclass attribute must be explicitly invoked from the corresponding original attribute via inner. The semantics reveals that the Beta inheritance hierarchy grows in the opposite direction from the Smalltalk hierarchy.

1.9 Inheritance in Flavors

Flavors, an object-oriented extension to Lisp, introduced the concept of multiple inheritance. There are two major problems in multiple inheritance: resolution of conflicts in what is inherited, and management of duplicate functionality from repeated ancestors. Flavors introduced method combination to resolve conflicts, and linearization to remove duplicate ancestors. Though these features do solve these problems; but they do so only at the cost of reduced compositionality. However, one of the more common uses of linearization and method combination, the *mixin convention*, is dependent upon exactly this non-compositionality. More compositional formulations for mixins are proposed.

1.10 Related Work

The most widely accepted analysis of inheritance is Cardelli's explanation of inheritance as a subtype relation [6]. Yet it is easy to show that subtyping is insufficient to explain the mechanism of inheritance in object-oriented languages. On the other hand, the subtype relation is a direct consequence of the model of inheritance introduced here. Thus the inheritance model presented here explains the presence of Cardelli's subtype relation, while subtyping alone is insufficient to explain inheritance.

The first full denotational semantics of a language with inheritance was presented by Kamin [14] almost twenty years after the concept was introduced. This unusually long delay is an indication of the novelty of inheritance, and its incompatibility with traditional techniques in denotational semantics. No other complete denotational semantics exists for a language with inheritance, though there are several attempts [22] [31] [32]. Kamin's semantics, though a fairly accurate description of Smalltalk, is not compositional in its treatment of inheritance.

Essentially the same model of inheritance presented in this thesis was developed independently by Reddy [23]. The model was only illustrated for Smalltalk, and was not developed to the degree of generality present here.

Chapter 2

The Inheritance Model

This chapter motivates an interpretation of inheritance as a mechanism for incremental programming in the presence of self-reference, and develops a formal model for this mechanism using techniques from traditional fixed-point semantics. Incremental programming is the definition of new program units by specifying how they differ from existing ones. Incremental programming differs from structured programming, which also involves hierarchical construction of programs, in that incremental programming allows modification of existing program units, rather than simply referencing or using them as they are. Thus incremental programming provides a new form of component reuse [3], useful when an existing component almost satisfies current requirements; incremental programming allows the component to be changed slightly rather than entirely rewritten.

Inheritance, as a mechanism for incremental programming, is necessary because of a subtle interaction between the process of modification and self-reference in the original structure. To support this claim, the concepts of modification and self-reference are investigated separately, and then their interaction illustrated. This leads to an intuitive explanation of inheritance and a formal model.

2.1 Modification

Modification, in the everyday sense, is a process of destructive change. When working with information structures, which can be freely copied and reused by any number of clients, modification takes on a less destructive character, and is expressed by a process of derivation. Derivation of a modified structure is achieved by placing a new structure between the original structure and its clients. This mediating structure provides new behavior in some cases and uses the original behavior in other situations. The modification is non-destructive in that the original only appears to be changed. Derivation is illustrated by the following picture, in which arrows represent invocation or reference. The client invokes the modification, M, and the modification invokes the original, P.

$$\begin{tikzcd}\text{client} \arrow[r] & \boxed{M} \arrow[r] & \boxed{P}\end{tikzcd}$$

Derivation

Perhaps the simplest and most general mechanism for derivation is function application. Under this interpretation M is a function that computes some new structure when applied to the original structure. Thus $M(P)$ would represent the derived value. This model is useful for its generality, but it is perhaps too general. Since M can be any function at all, little can be said about what kind of modifications it is making. A more structured analysis of modification is developed below.

2.1.1 Records

It is useful to formalize the notion of modification by introducing records. A record is an association of names to values, and is a very general representation for compound structure. Records are useful because they allow modification to be described as a collection of attributes to be added or replaced to achieve the modification. Records are also useful for more specific discussions of inheritance in object-oriented languages, because they are a good representation for objects.

A *record* is a finite mapping from labels to values [6]. The record concept used here is essentially equivalent to modules [2], environments [18], labeled products, etc. However, a record may simply be viewed as a collection of named attributes. A field name that does not have a value is assumed to be mapped to the undefined value $\perp$.

The association of a label to a value is called a *component* of the record. A record with labels $x_1, \ldots, x_n$ and values $v_1, \ldots, v_n$ is written

$$[x_1 \mapsto v_1, \ldots, x_n \mapsto v_n]$$

The value associated with a label x in a record m is $m(x)$; the notation $m.x$ will also be used occasionally.

The domain of a record is the set of labels it defines:

$\mathrm{dom}(m) = \{x | m(x) \neq \perp\}$

Restriction of a record m to the set of attributes X is expressed by $m|X$:

$m|X = \lambda x\,.\mathbf{if}\ x \in X\ \mathbf{then}\ m.x\ \mathbf{else}\ \perp$

2.1.2 Record Combination

The modification of records is formalized as *record combination.* It is useful to view the original structure and the modifications as both being records, and their combination represents the modifications acted out upon the original. A combination of two records is a record whose components are taken from two other records. A *record combination function* is a binary function on records that computes the combination of its arguments. Record combination functions are not necessarily commutative.

For example, the records $[x_1 \mapsto v_1, x_2 \mapsto v_2]$ and $[x_2 \mapsto v_2', x_3 \mapsto v_3]$ may be combined in a number of ways to produce a new record. One possible combination is: $[x_1 \mapsto v_1, x_3 \mapsto v_3]$

Two records *conflict* on a label if that label is defined in both records. The conflicting labels of two records m_1 and m_2 are $\mathrm{dom}(X) \cap \mathrm{dom}(Y)$. In the example above, x_2 is the only label in conflict. The definition of conflict is syntactic, in that it only concerns what labels are defined by the records, and not their values. Semantic conflict is also useful: two records semantically conflict on a label if they give different values to the label. However, this assumes that it is possible to compare values for equality, which is not always a theoretically nor practically sound assumption.

The primary issue in the definition of a combination function is how conflicts are resolved. The basic approaches are to prohibit conflicts, to make an asymmetric choice, or to compose the values in some way. The final option depends to a large degree on the type of values in the record. In the example above, the conflict on x_2 is resolved by omitting the conflicting label from the resulting combination.

Record combination functions will be denoted by $\oplus_\star$ where $\star$ is a conflict resolution function, which is applied to the pair of values for each label in conflict. Given records m and n, record combination is defined as follows:

		$s \in X - Y$	$s \in X \cap Y$	$s \in Y - X$
Strict	$(m \oplus_{\perp} n)(s)$	$m(s)$	$\perp$	$n(s)$
Left preferential	$(m \oplus n)(s)$	$m(s)$	$m(s)$	$n(s)$
Right preferential	$(m \oplus_{r} n)(s)$	$m(s)$	$n(s)$	$n(s)$
Composing	$(m \oplus_{\circ} n)(s)$	$m(s)$	$m(s) \circ n(s)$	$n(s)$
Applicative	$(m \oplus_{.} n)(s)$	$\perp$	$m(s)(n(s))$	$n(s)$

Table 2.1: Record combination functions.

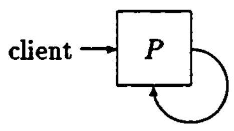
$$m \oplus_{\star} n = \lambda s\,.\begin{cases} m(s) & s \in \mathrm{dom}(m) - \mathrm{dom}(n) \\ n(s) & s \in \mathrm{dom}(n) - \mathrm{dom}(m) \\ m(s) \star n(s) & \textit{otherwise} \end{cases}$$

Various candidates for $*$ include $\perp$, the function that always returns bottom, $l = \lambda ab\,.\,a$ and $r = \lambda ab\,.\,b$ which select their first and second argument respectively, and $\circ$ for function composition.

Information on combination functions is summarized in Table 2.1. Wherever $\oplus$ appears without subscript it represents $\oplus_l$. Other combination functions are of course possible, especially if more complex structures are contained in the records.

2.2 Self-Reference

Self-reference occurs when a structure is defined in terms of itself. This technique is used frequently in programming, in the form of recursive procedures, functions, data types, etc. In the picture below, a self-referential structure is shown as a 'black box' with an arrow pointing from inside to the outside of the box; it accesses itself just like any other client. Self-reference is a form of invocation just like client invocation:



Self-reference

2.2.1 Fixpoint Semantics

The fixpoint semantics of recursive programs, developed by Scott[24], provides the mathematical setting for the inheritance model. Thorough introductory explanations are given by Stoy[27], Gordon[12], and Scott[24]. To illustrate the use of fixpoints for the analysis of recursive programs, consider the following definition of the factorial function:

$fact = \lambda n\,.\,\mathbf{if}\ n = 1\ \mathbf{then}\ 1\ \mathbf{else}\ n \times fact(n-1)$

The identifier *fact* is equated with a function definition in which the *fact* appears. In the body of the definition, *fact* is a bound variable that represents the value of the complete definition. Thus the definition is self-referential. The use of the symbol *fact* to represent self-reference is just a syntactic convention. The following definition uses an alternative notation (common in object-oriented languages) in which self-referential is represented by a special symbol, self:

$fact = \lambda n\,.\,\mathbf{if}\ n = 1\ \mathbf{then}\ 1\ \mathbf{else}\ n \times \mathsf{self}(n-1)$

The intent of such definitions is to specify the meaning, or denotation, of the identifier *fact*. This denotation is a function. But there is no guarantee that any function satisfying the above equation exists, and even less that there is a unique one. This problem is solved by fixpoint analysis, which indicates how to construct the denotation of these self-referential definitions. Using fixpoint techniques, the recursive definition is transformed into a non-recursive form. Within an appropriate domain, the value of this non-recursive definition is guaranteed to exist, and is also unique.

First, the body of the function is converted into an explicit abstraction, or function, in which the parameter self is substituted for *fact*:

$FACT = \lambda \mathsf{self}\,.\,\lambda n\,.\,\mathbf{if}\ n = 1\ \mathbf{then}\ 1\ \mathbf{else}\ n \times \mathsf{self}(n-1)$

FACT is a *functional*, or mapping from functions to functions; its definition is not recursive, because '*FACT*' does not appear its body. The formal parameter self represents the function to call in order to compute the factorial of numbers less than n, if needed. The original definition of *fact* may now be given in terms of *FACT*:

$fact = FACT(fact)$

But now *fact* is defined as a value that is unchanged when *FACT* is applied. Such a value is called a *fixed point*, or *fixpoint*, of *FACT*. Under certain conditions, it is possible to compute a unique *fixpoint*, the *least fixpoint*, of any function by using the *fixpoint function*, fix. The fixpoint function has the property that if $f = \text{fix}(F)$, then $F(f) = f$. The conditions which must hold for fix to work are almost always satisfied in the case of functionals derived from program definitions.

Although fixpoint semantics is necessary for the analysis of recursive definitions that produce infinite structures like the factorial function, it also works for degenerate recursive definitions that specify finite structures, which could just as easily be defined without recursion.

To illustrate, assume a pair constructor $\langle l, r\rangle$ and pair selectors *left* and *right*. The following definition is self-referential but not essentially recursive:

$p = \langle 3, \mathit{left}(p) + 1\rangle$

Clearly, $p = \langle 3, 4\rangle$. The fixpoint analysis of this definition proceeds in the same manner as for *fact*: first, a functional is defined in which the recursive reference is an explicit parameter:

$P = \lambda \mathsf{self}\,.\,\langle 3, \mathit{left}(\mathsf{self}) + 1\rangle$

Then $p = P(p)$ and $p = \text{fix}(P)$. It is easy to verify that $p = \langle 3, 4\rangle$ is the only pair for which $\langle 3, 4\rangle = P(\langle 3, 4\rangle)$.

The term 'recursive' might be expected to apply only when the fixpoint function is required, however, it is not always clear whether there is an equivalent value expressed without fixpoints. Thus there is little difference between recursion and self-reference. The term 'recursive' will be used primarily to describe a value defined by a fixpoint. 'Self-referential' has a more syntactic connotation, in describing a definition that refers to the what is being defined.

Technically, the notion of 'self-reference' applies only to *definitions*, not the denotations being defined, since denotations do not have internal, hidden structure. For example, it is the definition of factorial that is self-referential, not the factorial function itself (which is, for example, a flat collection of ordered pairs). Analogously, the denotation p above is simply a pair of integers.

However, the use of self-reference in a definition can express important dependencies in the value being defined. These dependencies are often transferred directly to the computational implementation of the definition. In the case of infinite structures like the factorial function, the only possible representation is as a self-referential procedure. As such, the value itself may reasonable be considered self-referential if it was defined self-referentially.

2.2.2 Generators

The concept of a *generator* is introduced to refer to the functions whose least fixpoints specify the meaning of recursive definitions. Introducing a special name for the arguments of the fixed-point operator is justified by the special conditions which are imposed on a function in that context.

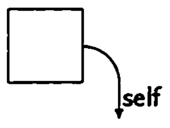
Definition 1 *A function intended to specify a fixpoint, whose formal parameter represents self-reference, is called a* generator.

The functionals *FACT* and *P* in the factorial and pair examples above are generators.

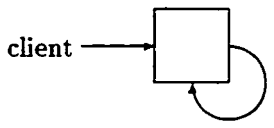
Generators will often be expressed with the variable self or s as the formal parameter. Thus a generator has the form

$G = \lambda \mathsf{self}\,.\, \mathit{body}$

where self may occur freely in *body*. Intuitively, self-reference is 'unbound' in a generator, as illustrated on the left below, while self-reference in its fixpoint is connected back to the generator, as illustrated to the right.



Generator



Fixpoint

Any function can be considered as a generator. It is the context of use within the fixed-point operator that motivates calling them generators. Thus it is impossible to distinguish generators from other functions except according to the use for which they are intended.

To illustrate consider the following record generator representing a pair of values where the second depends upon the first.

$G_1 = \lambda \mathsf{self}.\,[\mathsf{base} \mapsto 7, \mathsf{square} \mapsto \mathsf{self.base} * \mathsf{self.base}]$

The fixpoint of this generator is the record

$m_1 = \mathrm{fix}(G_1) = [\mathsf{base} \mapsto 7, \mathsf{square} \mapsto 49]$

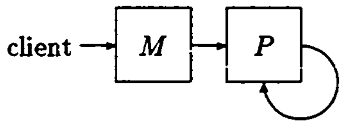
The value of attribute square in m_1 is $m_1(\mathsf{square}) = 49$.

Although any function is a generator, not every generator has a fixpoint. If the generator makes use of itself in ways that are incompatible with the structure it creates, then the generator fails to specify a valid value. Conditions governing when a generator has a fixpoint are developed in Section 4.1.

A generator may play two roles: (1) it may be fixed to specify a recursive object, and (2) it may be modified to define a new generator. The first role is the traditional use of generators, as they occur in fixpoint semantics. The second role is unique to inheritance.

2.3 Inheritance

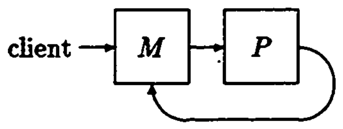
The motivation for inheritance is an interaction between self-reference and modification. To see why, consider a naive attempt to modify a self-referential value by derivation. This situation is illustrated below:



Naive derivation and self-reference

It is clear that this derivation has not simulated the effect of destructive modification, becuase self-reference be used to refer to parts of the original structure that should have been modified.

Since non-destructive modification by derivation allows the modifications to be separate from its original, special measures must be taken to ensure that self-reference is handled properly. In short, self-reference in the original must be changed to refer to the modifications. This is the mechanism of inheritance:



Inheritance

This interaction illustrates why a special mechanism is required to achieve modification of self-referential structures: because self-reference must be updated to complete the derivation. This process is an essential aspect of inheritance in object-oriented languages. Most programming languages cannot express this form of derivation, and even object-oriented languages provide it only for certain kinds of structures. Thus inheritance represents a truly novel and fundamental mechanism for constructing programs.

2.3.1 A Formal Model of Inheritance

The interpretation of inheritance presented above is formalized using generators. The essential observation is that the manipulation of self-reference can be modelled as an operation on generators.

Definition 2 Inheritance *is the derivation of a new generator from one or more existing generators, such that the formal parameter of the derived generator (representing self) is passed to all of the inherited generators.*

Examples of inheritance are given in Chapter 3.

The new generator *inherits* from the original generators. The original generators are called *parents.* The derived generator is called the *child.* When there is no chance of ambiguity, the corresponding fixpoints may also be called the parent and child. The child is derived by *single inheritance* if it only has one parent, or *multiple inheritance* if there are more than one.

Since generators are closely related to self-referential definitions the effect of inheritance can be understood as an operation on definitions. In this context, inheritance corresponds to textually embedding an existing definition inside a new definition, when using the syntactic convention of representing self-reference by the keyword **self** Since the same identifier is used to represent self-reference in the inherited definition and the definition in which it is embedded, self-reference is shared between them. This interpretation served as the original definition of inheritance for the language Simula[8].

The effect of inheritance can also be understood at the level of the values defined by self-referential definitions. In this case the changes effected by inheritance represent pervasive modification. The modification is *pervasive* because it affects every reference to the original value, including self-reference within the value itself.

One significant consequence of the process of inheritance is that the fixpoint of a child must resemble the fixpoint of its parent generator. This is because self-reference in the parent, which originally referred to the parent fixpoint, is changed to refer to the child fixpoint during inheritance. Since both parent and child fixpoints must satisfy this same self-reference, they must resemble each other at least to the degree specified by the parent self-reference. This consequence of inheritance is investigated further in Chapter 4.

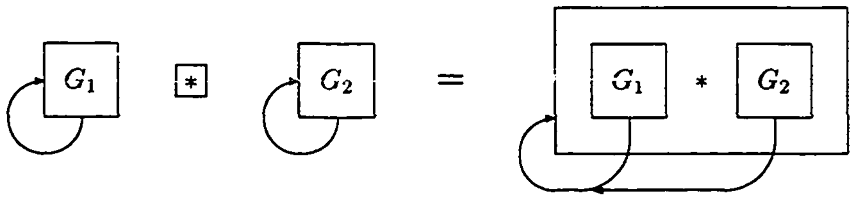
2.3.2 Distributive Combination

The mechanism of *distributive combination* is introduced as a concise notation for expressing the manipulation of self-reference that is necessary to achieve inheritance. Given a function for combination of values, the corresponding function for combining generators is provided by distributive combination: a common binding for self-reference is distributed to both generators, and their value combined. The result is a new generator that inherits from both of the original ones.

Any binary operator can be converted into a distributive form. Given a binary operator $*$, the *distributive operator* $\boxed{*}$ is a binary operator on generators, defined by supplying the same self-reference to the to generators, and combining their results with $*$. The distributive version of $*$ is defined as:

$$G_1 \boxed{*} G_2 = \lambda s \, . \, G_2(s) * G_2(s)$$

This 'boxed' form $\boxed{*}$ of an operator $*$ is called distributive because $(G_1 \boxed{*} G_2)(x) = G_2(x) * G_2(x)$. The distributive combination of two generators is illustrated below:



The *distribution function* $\boxed{}$ converts binary operators to distributive operators, and may be viewed as a ternary function, taking a binary function and two arguments, and constructing the distributive combination of the arguments using the binary function. Thus $\boxed{}$ is a general mechanism for expressing distributive operators.

To illustrate distributive record combination, the generator G_1 defined on Page 14 is combined with another (trivial) generator which defines a different value for base:

$G_2 = \lambda \mathsf{self}.\, [\mathsf{base} \mapsto 2]$

The distributive combination of G_2 and G_1, using the preferential combination function $\oplus$ (the same as $\oplus_l$) is:

$$\begin{aligned} G_1 \boxed{\oplus} G_2 &= \lambda \mathsf{self}.\, G_2(\mathsf{self}) \oplus G_1(\mathsf{self}) \\ &= \lambda \mathsf{self}.\, [\mathsf{base} \mapsto 2] \oplus [\mathsf{base} \mapsto 7, \mathsf{square} \mapsto \mathsf{self.base} * \mathsf{self.base}] \\ &= \lambda \mathsf{self}.\, [\mathsf{base} \mapsto 2, \mathsf{square} \mapsto \mathsf{self.base} * \mathsf{self.base}] \end{aligned}$$

The fixpoint of the resulting generator is $[\mathsf{base} \mapsto 2, \mathsf{square} \mapsto 4]$.

Distributive operators compose naturally, allowing for a single self to be distributed to all constituent expressions. For example, $(a \boxed{*} b) \boxed{+} c$ represents $\lambda s.(a(s) * b(s)) + c(s)$.

Distribution can also be nested to distribute a second parameter over the arguments:

$$\begin{aligned} G_1 \boxed{\boxed{\cdot}} G_2 &= \lambda s.(G_1(s) \boxed{\cdot} G_2(s)) \\ &= \lambda s.\lambda t.G_1(s)(t) \cdot G_2(s)(t) \end{aligned}$$

2.4 Varieties of Inheritance

2.4.1 Wrapping

Wrapping is a general form for inheritance that derives from handling self-reference within the interpretation of modification as function application. The concept of a *wrapper*[1] is introduced to describe a form of modifying function that is applied in such a way that it can refer to the result of the modification. Wrappers are the basic general form for modifying self-referential structures.

[1]The term 'wrapper' comes from Flavors, where it represents a kind of method that is combined in a way similar to what is described here. The notion of wrapper used in this thesis is perhaps more directly analogous to Flavors 'mixins', as described in Section 10.5, but 'mixin' does not have the right connotations.

Definition 3 *A* wrapper *is a function designed to modify a self-referential structure in a self-referential way; it has two parameters, one representing self-reference and the other representing the super-structure being modified.*

Thus a wrapper is a function of the from λself.λsuper.*body* , where **self** and **super** may occur free in *body*.

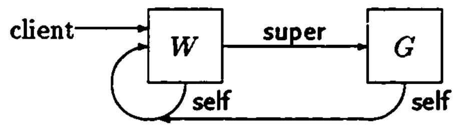
The application of a wrapper to a generator involves binding together self-reference in the wrapper and the generator, and then applying the wrapper modification to the value of the generator. Given a generator G and a wrapper W a new generator $W \boxed{\cdot} G$ is derived using the *distributive application function* $\boxed{\cdot}$, defined as follows:

$W \boxed{\cdot}\, G = \lambda \mathsf{self}.\, W(\mathsf{self})(G(\mathsf{self}))$

The wrapper application function $\boxed{\cdot}$ is the distributive version of application, where $\cdot$ is used to express application: $f \cdot x = f(x)$. Written out as a lambda expression, $\boxed{\cdot}$ is the S combinator form algebraic models of the λ-calculus:

$\boxed{\cdot} = \lambda a.\lambda b.\lambda s.a(s)(b(s))$

Note that the result of this application is a generator. The effect of applying the wrapper W to a generator G is illustrated below. The curved arrows represent self-reference. Self-reference in W and G are bound together though the variable self, signified by the joining of the arrows out of W and G. The arrow from W to G represents the application of W to G.



2.4.2 Record Inheritance

Wrapper record inheritance uses record wrappers to provide an explicit format for modification of record generators. A wrapper specifies changes to its parent as additional or modified components, and has explicit access to the original attributes in the parent.

The two generators that are combined during wrapper inheritance are the wrapped parent and the original parent. In this way any changes specified by the wrapper may replace corresponding attributes in the original parent. All other components of the the

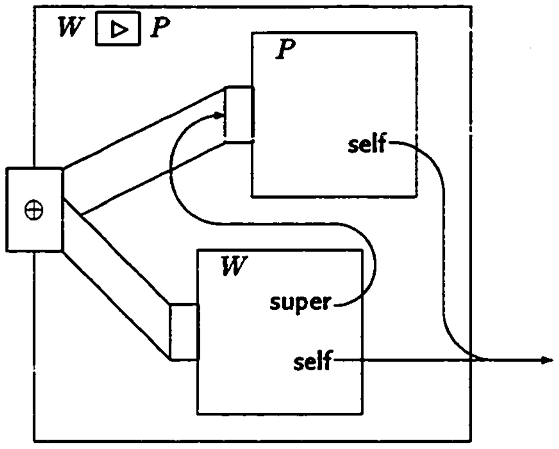


Figure 2.1: Record inheritance with combination.

parent be simply transferred to the child. The structure of records as compound objects makes this refinement of the notion of modification possible. The combination function chosen determines what kind of changes the wrapper is allowed to make.

A *record wrapper* is a binary function on records. Its first argument represents self-reference, and the second argument represents the record being modified. A record wrapper specifies the self-referential components to be combined with the parent record.

A record wrapper is applied to a record generator to produce a new record generator. The wrapper uses the resulting record and the parent record, and resulting record is the combination of the wrapper and the parent. Record wrapper application with combination function $\oplus$ is defined by the infix operator $\boxed{\triangleright}$ as follows:

$$W \boxed{\triangleright} P = (W \boxed{\cdot} P) \boxed{\oplus} P$$

The inheritance function is the distributive form of the operator $w \triangleright p = (w \cdot p) \oplus p = w(p) \oplus p$ which might be used for modifying a record. The box operator distributes uniformly over the binary operators for application and addition (see Section 2.3.2).

With this construction it is possible for the wrapper to access all of the components of the parent definition. The effect of this definition is illustrated in Figure 2.1. Note that the multiple occurrences of P do not indicate that the parent is 'instantiated' twice. On the contrary, P is simply an variable which denotes the unique value of the parent generator, and it is this value which is referred to twice.

An alternative construction that is useful in some cases is to allow

Preferential record inheritance is a simplification of wrapper inheritance that uses self-reference to provide access to the unmodified parent attributes. The basic mechanism of preferential inheritance is simply the combination of two two record generators G_1 and G_2 by the (left) preferential combination is $G_1 \boxed{\oplus} G_2$. Attributes from G_1 take precedence over those in G_2. This asymmetry gives rise to a difference in roles for the two inherited generators: G_2 often represents the base or original attributes, while G_1 represents changed or updated attributes.

2.4.3 Selective Record Inheritance

Selective inheritance is a restricted form of record combination inheritance in which the modification components can only access the corresponding component in the parent, not arbitrary components. A different kind of modification record is used during selective inheritance: it contains functions that are composed with the corresponding functions in the parent. A selective modification is a record of transformation functions on a domain. The modification associates with each label a function on the corresponding parent attribute, and is suitable for combination with $\oplus_\circ$. Selective record inheritance is defined by the infix form $M \boxed{\oplus_\circ} P$.

Selective inheritance is used in the semantics of Beta given in Chapter 9.

2.5 Multiple Inheritance

Multiple inheritance is a generalization of single inheritance which allows multiple parents to be involved in the construction of the child. The definition of wrappers must be extended to provide for multiple inheritance. In this formalization, the wrapper is responsible for resolution of all conflicts among its parents, as well as the explicit transfer of their properties into the result. An n-wrapper W is a wrapper that uses n parents to construct a child.

The parents are placed in a tuple of size n. The manipulation of tuples containing generators is facilitated by defining the application of a tuple as a function: When a tuple is applied to an argument, the value is a tuple consisting of each element applied to the argument:

$$\langle f_1, \ldots, f_n \rangle(a) = \langle f_1(a), \ldots, f_n(a) \rangle$$

Multiple inheritance is similar to single inheritance, except the parent type of the wrapper is taken as a tuple. A child C of multiple inheritance of parents $G_1, \ldots, G_n$ and wrapper W is:

$$\begin{aligned} C &= W \boxed{\cdot} \langle G_1, \ldots, G_n \rangle \\ &= \lambda \mathsf{self}.\, W(\mathsf{self})(\langle G_1(\mathsf{self}), \ldots, G_n(\mathsf{self}) \rangle) \end{aligned}$$

This characterization of multiple inheritance, while not complete, provides a framework in which more complex forms may be studied. The wrappers themselves are the focus for further development, for they are where issues relating to resolution of conflicts among parents or automatic combination of parents must be addressed. Development of more sophisticated inheritance mechanisms within this framework requires the addition of a layer of structure within the self-referential values whose generators are being derived.

2.5.1 Strict Multiple Record Inheritance

Strict multiple record inheritance provides strict combination of multiple parent generators followed by record wrapping. To perform multiple inheritance automatically, the parents are first combined into a single generator. In this case, the parents are combined strictly by $\oplus_\perp$ so that conflicting symbols are removed. The result of this combination is then wrapped by a record wrapper, which is applied to the original list of parents, rather than their strict combination, to allow explicit access to each parents' original attributes. Since conflicts among the parents are converted into error values, errors will be transferred to the resulting generator unless they are over-ridden by the wrapper. A child C of multiple inheritance of parents $G_1, \ldots, G_n$ and wrapper W is:

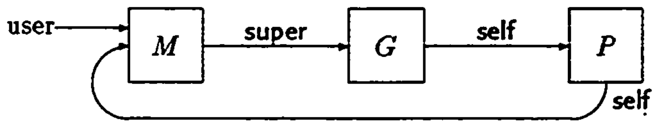
$$W \boxed{\rhd^n} \langle P_1, \ldots, P_n \rangle = (W \boxed{\cdot} \langle P_1, \ldots, P_n \rangle) \boxed{\oplus} (P_1 \boxed{\oplus_\perp} \cdots \boxed{\oplus_\perp} P_n)$$

Obviously, other constructions for multiple inheritance are possible.

2.6 Generalized Inheritance

The restriction that self-reference must be shared among all generators during inheritance may be relaxed to allow more general forms of 'inheritance'. In the most general sense, a derived generator is simply a function of the original generator. For example, given the generator G and some appropriate function M, a new generator G' may be derived by $G' = M(G)$. This proposal does not immediately indicate applications, since meaningful operations on generators do not seem likely. Thus it seems reasonable to include the binding of self-reference in the derivation mechanism, as in the forms of inheritance defined in previous sections which require that self-reference be passed intact directly to the parent generator. However, an intermediate is possible between these strict forms and the generalization presented above.

A potentially useful generalization of inheritance is defined by allowing the self-reference passed to the inherited generators to be modified, as well as the exterior modification described above. This is achieved by composing two functions around the parent generator. For example, given two modification functions M and P, and a generator G, a generator may be defined by $G' = M \circ G \circ P$. In this case the actual parameter of G, which it interprets as self, is in this case a modification the self-reference of the derived generator. Thus all self-reference within G will invoke a modified self. Such internal pervasive modification is illustrated below:



It may prove useful to use this form to adapt the interface through which the original makes self-references so that it conforms to derived generator's external interface. This technique is illustrated in Section 3.2. It may also be useful for performing renaming of attributes, because the internal use of names must be changed so that self-reference to the renamed attribute will also be renamed.

Chapter 3

Examples of Inheritance

3.1 Object Inheritance

The traditional use of inheritance is for incremental derivation of the methods used by classes of objects. The objects illustrated in this section are a simplification of the objects in full object-oriented languages; they are *immutable* in that they do not have changeable state. A more complete analysis of inheritance for mutable objects is given in the chapters on the denotational semantics of object-oriented languages.

Objects are naturally modelled as records containing functions which represent methods or operations. These methods constitute a collection of mutually recursive definitions. A class is a constructor that creates objects with similar methods. Record generator combination, as introduced in Section 2.4.2, is used to achieve inheritance.

The following example illustrates how fixpoints can be used to describe the behavior of objects. The example involves a simple class of 'points'. Points have x and y components to specify their location. The distFromOrig method the computes their distance from the origin. Finally, closerToOrg is a method that takes another point object and returns the one that is closer to the origin.

```
class Point(a, b)
    method x = a
    method y = b
    method distFromOrig = sqrt(self.x$^2$ + self.y$^2$)
    method closerToOrg(p) = (self.distFromOrig < p.distFromOrig)
```

The class Point can be represented as a function that creates objects. Self-reference in Point may be explained using fixpoint techniques. A function Point is defined to construct points and supply them with appropriate functions to represent methods. Since points are self-referential, they are explained as the fixpoint of the "generator" of the methods. Point is function that takes the coordinates of the new point and returns a generator, whose fixpoint is a point.

```
Point(a, b) = λself.
    [ x ↦ a,
      y ↦ b,
      distFromOrig ↦ sqrt(self.x² + self.y²),
      closerToOrg ↦ λp.(self.distFromOrig < p.distFromOrig)]
```

A point at $(3,4)$ is created by

$p = \text{fix}(\text{Point}(3,4))$

This point is equivalent to the following record:

```
p = [ x ↦ 3,
      y ↦ 4,
      distFromOrig ↦ 5,
      closerToOrg ↦ λp.(5 < p.distFromOrig)]
```

The closerToOrg function takes a single argument which is assumed to be a point. Actually, all that is required is that it be a record with a distFromOrig method, whose value is a number.

Inheritance allows a new class to be defined by adding or replacing methods in an existing class. For example, the Point class may be inherited to define a class of circles. Circles have a different notion of distance from the origin. This definition gives only the differences between circles and points:

```
class Circle(a, b, r)
    inherit Point(a, b)
    method radius = r
    method distFromOrig = max(super.distFromOrig - self.radius, 0)
```

This form of inheritance can be modelled as generator modification. There are three aspects to this process: (1) the addition or replacement of methods, (2) the redirection of self-reference in the original generator to refer to the modified methods, and (3) the binding of **super** in the modification to refer to the original methods.

The modifications effected during class inheritance are naturally expressed as a record of methods to be combined with the inherited methods. The modifications are not simply are record, however, because they may be defined in terms of the original methods (via **super**). In addition, the modifications may make self-references to the resulting structure. Thus a modification is naturally expressed as a function of two arguments, one representing **self** and the other representing **super**, that returns a record of locally defined methods. Such functions were called *wrappers* in Section 2.4.1. A wrapper contains just the information in the subclass definition:

$$\begin{array}{l} \text{CircleWrapper} = \lambda\, a, b, r\,. \\ \quad \lambda\,\text{self}\,.\,\lambda\,\text{super}\,.\,\left[\begin{array}{l} \text{radius} \mapsto r \\ \text{distFromOrig} \mapsto \max(0,\ \text{super.distFromOrig - self.radius}) \end{array}\right] \end{array}$$

The other aspect of inheritance that must be formalized is the change of self-reference in inherited methods. The methods to be inherited are contained in a generator, a function whose argument is used for self-reference in the methods. The result of inheritance should be a new class definition, that is, a new generator.

The generator which represents the class Circle can now be defined by wrapper application of CircleWrapper to Point:

$$\text{Circle} = \lambda\, a, b, r\,.\,\text{CircleWrapper}(a, b, r)\ \boxed{\triangleright}\ \text{Point}(a, b)$$

The resulting generator is equivalent to the following one, which is exactly what one might write if circles had been defined without using inheritance:

$$\begin{array}{l} \text{Circle} = \lambda\, a, b, r\,. \\ \quad \lambda\,\text{self}\,.\,\left[\begin{array}{l} \text{x} \mapsto a, \\ \text{y} \mapsto b, \\ \text{radius} \mapsto r \\ \text{distFromOrig} \mapsto \max(0,\ \text{sqrt}(\text{self.x}^2 + \text{self.y}^2)\ \text{- self.radius}), \\ \text{closerToOrg} \mapsto \lambda\, p\,.(\text{self.distFromOrig} < p\text{.distFromOrig}) \end{array}\right] \end{array}$$

Note that distFromOrig has changed in such a way that closerToOrg will use the notion of distance for circles, instead of the original one for points. Thus inheritance has achieved a consistent modification of the point class.

3.2 Function Inheritance

An additional level of inheritance is required when the constructor function itself is recursive. In the previous example, the parent class Point was simply combined with some new definitions to produce the child Point. When Point itself is a recursive function, additional mechanism is needed. Consider a class of points with a move method:

$$
\begin{aligned}
\mathsf{MovingPoint} \;=\; & \lambda\, x, y \,.\, \lambda \mathsf{self}. \\
& [\;\; \mathsf{x} \mapsto x, \\
& \quad \mathsf{y} \mapsto y, \\
& \quad \mathsf{move} \mapsto \lambda\, d \,.\, \mathrm{fix}(\mathsf{Point}(\mathsf{self.x} + d.\mathsf{x}, \mathsf{self.y} + d.\mathsf{y})) \\
& \quad \mathsf{distFromOrg} \mapsto \ldots \\
& \quad \mathsf{closerToOrg} \mapsto \ldots]
\end{aligned}
$$

In this example the constructor is recursive, in addition to self-reference in the points themselves. To illustrate inheritance in of recursive constructors, a class of moving circles is defined by analogy with the example in the previous section.

Since the constructor is recursive, inheritance should be used to derive a modified constructor. To perform inheritance, it is necessary to consider the generator of the Point constructor:

$$
\begin{aligned}
\mathsf{MovingPoint} \;=\; & \lambda \,\mathsf{make\text{-}self} \,.\, \lambda\, x, y \,.\, \lambda\, \mathsf{obj\text{-}self}. \\
& [\;\; \mathsf{x} \mapsto x, \\
& \quad \mathsf{y} \mapsto y, \\
& \quad \mathsf{move} \mapsto \lambda\, d \,.\, \mathrm{fix}(\mathsf{make\text{-}self}(\mathsf{obj\text{-}self.x} + d.\mathsf{x}, \mathsf{obj\text{-}self.y} + d.\mathsf{y})) \\
& \quad \mathsf{distFromOrg} \mapsto \ldots \\
& \quad \mathsf{closerToOrg} \mapsto \ldots]
\end{aligned}
$$

The two levels of recursion are identified by the bindings of make-self and obj-self. Inheritance of the constructor is important because points have the ability to make other points; a modified kind of point should call the modified constructor when it wants to make a version of itself. If the new constructor is not derived using inheritance then when a circle was moved the it would become an ordinary point. The wrapper for this example is the same as in the previous section.

Now the constructor for moving circles is going to have to take three arguments: x, y, and r. In the following definition, it is important to realize that MovingPoint takes as its first argument the function that moving point instances use to create new points. Since they should make moving circles instead of just moving points, the new constructor being defined is supplied. This definition is a first cut at the solution:

$$
\begin{aligned}
&\mathsf{MovingCircle} \;=\; \lambda\, a, b, r\,.\\
&\quad \mathsf{CircleWrapper}(a, b, r) \;\boxed{\triangleright}\; \mathsf{MovingPoint}(\mathsf{MovingCircle})(a, b)
\end{aligned}
$$

The only problem with this definition is that MovingPoint expects its argument to take two parameters, just x and y. However, MovingCircle requires three arguments, including the radius value. This is because the moving point generator only knows that the point is moving, it doesn't know anything about circles. Assuming that the new radius is the same as the old, the following definition adapts the recursive call so that the radus is supplied. In addition, it defines a new constructor generator, as is more proper:

$$
\begin{aligned}
&\mathsf{MovingCircle} \;=\; \lambda\, \mathsf{make\text{-}self}\,.\,\lambda\, a, b, c\,.\\
&\quad \mathbf{let}\ \mathsf{make\text{-}super} = \mathsf{MovingPoint}(\lambda\, a', b'\,.\,\mathsf{make\text{-}self}(a', b', r))\ \mathbf{in}\\
&\qquad \mathsf{CircleWrapper}(a, b, r) \;\boxed{\triangleright}\; \mathsf{make\text{-}super}(a, b)
\end{aligned}
$$

It would also be possible to have the radius change during a move, so that it would depend upon the new location, the distance moved, etc. In any case, this expression is equivalent to the following one, which is clearly what was desired:

$$
\begin{aligned}
&\mathsf{MovingCircle} \;=\; \lambda\, \mathsf{make\text{-}self}\,.\,\lambda\, a, b, r\,.\,\lambda\, \mathsf{obj\text{-}self}\,.\\
&\quad [\quad \mathsf{x} \mapsto a,\\
&\qquad\; \mathsf{y} \mapsto b,\\
&\qquad\; \mathsf{move} \mapsto \lambda\, d\,.\,\mathsf{fix}(\mathsf{make\text{-}self}(\mathsf{obj\text{-}self.x} + d\mathsf{.x}, \mathsf{obj\text{-}self.y} + d\mathsf{.y}, r))\\
&\qquad\; \mathsf{distFromOrg} \mapsto \ldots\\
&\qquad\; \mathsf{closerToOrg} \mapsto \ldots\\
&\quad]
\end{aligned}
$$

Upon careful examination, it is clear that this example illustrates the generalization of inheritance described in Section 2.6. This is because the new constructor generator does not pass self directly to its parent; it is modified by the adapting function that calls the modified constructor with the original radius.

3.3 Procedure Inheritance

Procedure inheritance allows the definition of modified forms of an existing procedure, such that recursion in the original procedure is redirected to invoke the modified form. Procedure inheritance is significant because it provides elegant solutions to problems which cannot be solved reasonably in traditional programming languages. Yet it is a simple matter to introduce inheritance into traditional languages, immediately increasing their expressive power.

Memoization is a good example of the power of procedure inheritance to solve a difficult problem. A function is memoized by converting it in to a procedure owning a table in which previously computed function values are stored. The procedure is used in place of the function; it computes function values on demand and stores them in the table, but simply returns any previously computed values. The solution is significant because the general memoization problem cannot be solved reasonably in conventional programming languages.

The difficulty of the general memoization stems from the problem of recursion in the function being memoized. A naive attempt at memoization by simply defining a procedure that invokes the function on demand fails when the function is recursive. Since a recursive function makes calls *directly* to itself and it will not take advantage of the memos containing stored results, when invoked by the naive memoization. Conventional programming languages are not able to express internal use of memos without radically changing the organization of the system. The most common way to make recursion in the function use the memos is to copy the function definition and edit its definition, intermixing the memoization with the original function behavior. This approach introduces unwanted redundancy and increases the maintenance costs of the resulting systems. Another choice is to rewrite the function to take a additional formal parameter representing the function to call to perform recursion. If the original function has the form

$F = \lambda x \ldots . F(e) \ldots,$

then the rewrite has the form

$$G = \lambda f . \lambda x . \dots f(f)(e) \dots,$$

and the original function call $G(a)$ is simulated by $G(G, a)$. This option amounts to simulating the fixed-point construction of specialization by self-application [10] and is closely related to delegation [17]. This technique is prone to errors that are hard to detect and often involves violating a language's type system, since it requires the use of self-application.

Using inheritance, the memoization of a function F is easily expressed:

```
private
    T = new Table
in
    F' = inherit F in fun(v)
        if has(T, v) then
            get(T, v)
        else
            set(T, v, super(v))
```

Function specialization allows for the evolution and modification of a recursive function without physically changing the original.

The difficulty of "memoization" is well-known [1, p. 69]:

> Memoising also presents another interesting challenge to the designers of functional programming languages. Ideally one would like to be able to define a higher order function that takes a function as argument and yields a more efficient version of the function as a result. It is easy enough to write down the rules for transforming the functions, but to implement the transformation requires access to the structure of the function.

They resort to copying code and modifying it textually. Although their "ideal case" is impossible to implement, it is possible to write a higher-order function that operates on the *generator* of a function to produce a more efficient version. It is significant that only limited access to the structure of a function is needed; the only access needed is to the *recursive* structure of the function.

3.4 Data Type Inheritance

The types found in programming languages like Pascal may also be constructed using inheritance, a possibility noted by Borning and Ingalls [4]. Data-structure inheritance allows an existing recursive data structure to be specialized, typically by adding fields to the record representing a node in the recursive structure. All levels of this data structure would be specialized, because recursive pointers would refer to the specialized definition. Consider the following tree type:

```
type Tree = record left, right : ↑tree end;
```

Integer trees inherit the structure of Tree and add a value field to each node:

```
type IntTree = inherit Tree &
                       record value : Integer end;
```

The combination function is Cardelli's & operator [7] for concatenation of two record types. The resulting definition of IntTree is equivalent to the following definition without inheritance, in which self-reference to Tree in the parent definition is changed semantically to IntTree in the child:

```
type IntTree = record left, right : ↑ IntTree;
                      value : Integer end;
```

Inheritance may be used to modify a function and the data structures on which it operates in parallel. A function that handles several different cases may be extended by inheritance to handle more cases. The new cases which the function handles may also be added by inheriting the original data structure definition. The result is a parallel specialization of data and functionality. For example, an evaluator of a simple expression language, where expressions are encoded in the following data structure is:

```
SumExp ::= Value of Integer | Sum of SumExp × SumExp
```

The function

```
SumEval : Exp →Integer
```

evaluates expressions in Exp as follows:

```
SumEval = fun(var t : SumExp)
    case t of
        Value : fun(i) i
        Sum : fun(t1, t2) SumEval(t1) + SumEval(t2)
    endcase
```

To add a product case to the structure of expressions, a new data structure is defined that inherits from SumExp:

```
ProdExp = inherit SumExp & Product of ProdExp × ProdExp
```

The evaluation function for ProdExp's is defined by inheriting the behavior of SumEval, and adding a case to handle products:

```
ProdEval = inherit SumEval by
    fun(var t : ProdExp)
    case t of
        Product : fun(t1, t2) ProdEval(t1) * ProdEval(t2)
        others: super(t)
    endcase
```

3.5 Hierarchy Inheritance

A more novel use of inheritance is in the derivation of modified hierarchies or other graph structures. The links between nodes in the graph are interpreted as self-references from within the graph to itself. By inheriting the graph and modifying individual nodes, any access to the original nodes will be redirected to the modified versions.

For example, in the context of object-oriented programming, a complete class hierarchy may be inherited, while deriving new definitions for some internal classes. The result of this inheritance is a modified class hierarchy with the same basic structure as the original, but with the total behavior of all classes modified that depend upon the classes that were explicitly changed. The resulting hierarchy may be grafted back into the larger structure.

This problem was first proposed by Henry Lieberman [17] in the form of a class hierarchy containing the definitions of a number of graphical shapes, which represent the corresponding planar regions, as illustrated in figure 3.1.

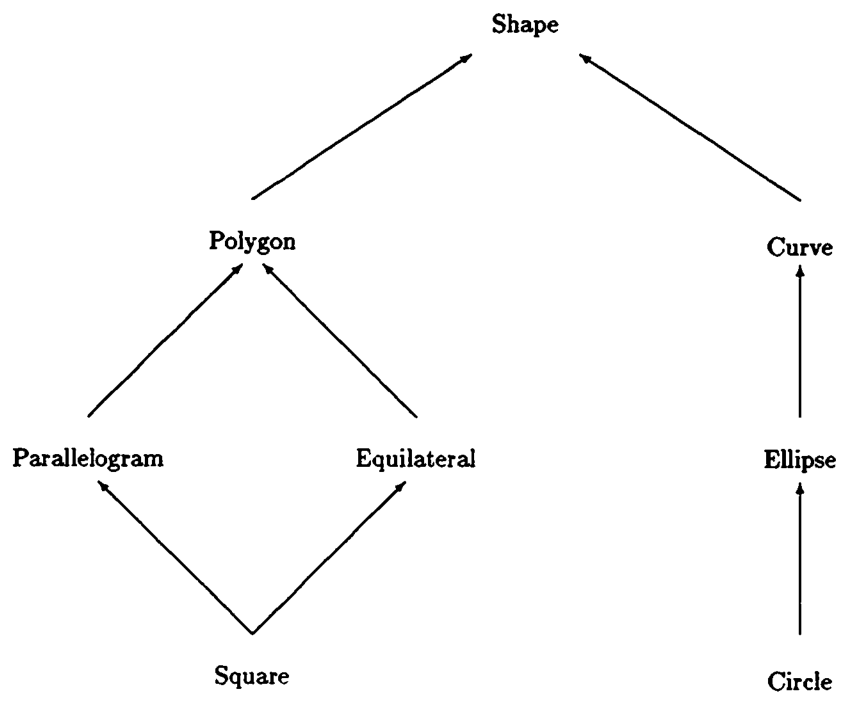


Figure 3.1: A shape hierarchy

A color display is introduced into the system and it becomes useful to have a similar shape hierarchy for colored shapes, which are similar to black-and-white shapes, but have some extra fields and methods added to handle color operations.

One solution to this problem is to edit the shape class and add the necessary definitions. Although this has the unfortunate property of destroying the black-and-white hierarchy, it might be possible to view black-and-white as a special case of the new color hierarchy.

The second solution depends upon multiple inheritance, and allows both color and black-and-white hierarchies to exist at the same time. The alternative, given a language with multiple inheritance, is to define a class *ColorShape* that inherits from *Shape* and then manually recreate the hierarchy by making a new class Color c that for each descendent c of *Shape*, such that Color c inherits both c and the color version Color p any parent p of c. (This brings up problems of duplication of ancestors, but this issue will be ignored here.) The resulting structure is a parallel hierarchy, shown in figure 3.2

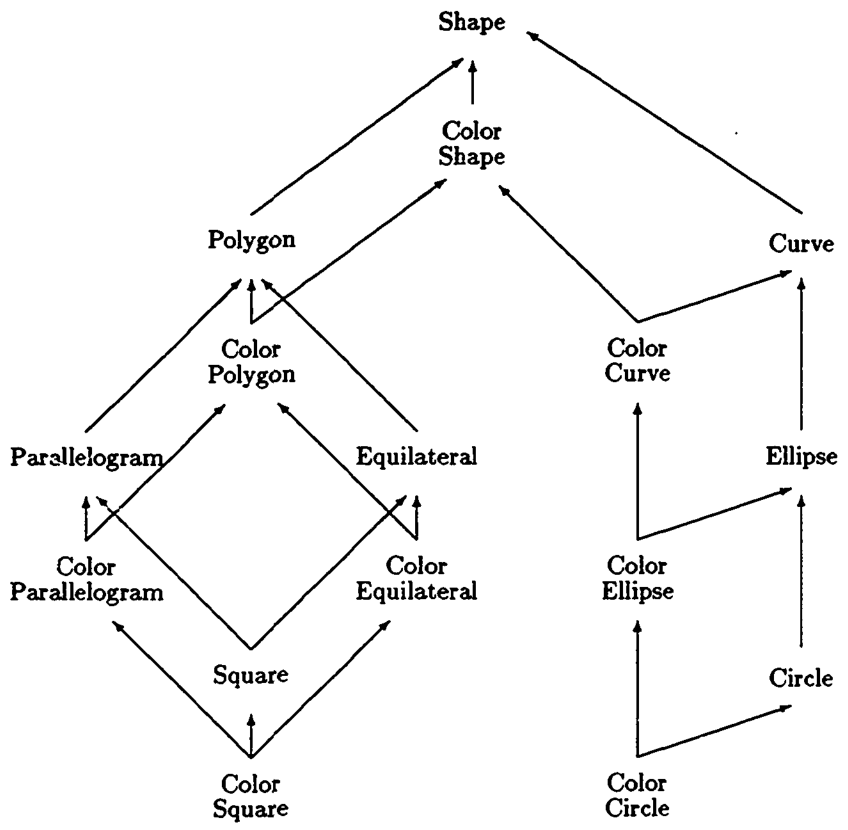


Figure 3.2: A derived shape hierarchy.

If multiple inheritance is not available, as in Smalltalk, then it is necessary to copy each class under shape.

A more elegant solution is to allow 'horizontal' inheritance of the entire shape hierarchy. What is needed is the ability to collect these classes up into a unit. This subhierarchy, called BW, is a mapping from class names to definitions; it is a class environment. When a class definition, like Polygon, specifies that it is a subclass of Shape, this should be interpreted as being a subclass of BW.Shape.

Given an appropriate combination function **with** the color hierarchy could then be defined as

```
hierarchy Color
inherit BW
```

```
with
    class Shape inherit super.Shape
        {additional features of color shapes}
```

The combination function on hierarchies would be designed to specialize each class in the parent hierarchy by the corresponding class definition in the hierarchy specialization. Thus Color.shape would automatically inherit BW.shape. The inheritance of sub-hierarchies depends upon the fact that recursive environments may be decomposed into a collection of smaller recursive bindings, which are then combined into a larger recursive binding.

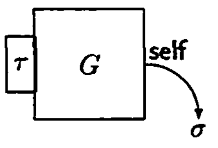
Chapter 4

Type Theory and Inheritance

4.1 Typing Generators

Examining the types of generators leads to an understanding of the external and internal interfaces of a self-referential definition, and the constraints on their use. Since the results of this application are useful for using inheritance, but have little significance for type theory, the presentation will be at an informal level. There is much more that can be said about the interaction between type theory and inheritance, but it is beyond the scope of this thesis, and will have to await future research.

As a first approximation, since generators are functions, they have types of the form $\sigma \rightarrow \tau$. The formal parameter of a generator represents self-reference, hence the type σ of the formal parameter represents the type of self-reference that the generator makes. The range τ of a generator is the type of result the fixpoint operation creates. The self-reference type σ represents the assumptions the generator makes about its fixpoint, because it is as a value in the self-reference type that the result of a generator refers to itself. In the illustration below, the result-type τ describes the external interface of the generator fixpoint, while the self-reference type σ describes the internal reference from within the definition to the external interface:



4.2 Generator Consistency

The typing of generators provides strong constraints governing the existence of fixpoints. As mentioned in Section 2.2.2, not all generators are valid specifications of a fixpoint. A generator fails to have a fixpoint if its references to itself do not match its result type. The typing constraints are defined in terms of subtyping [6, 19].

A generator is *consistent* if it has a fixpoint. The relationship between σ and τ determines whether a generator is consistent. If τ subtype σ, then any value of type τ can be used as a value in σ, and any generator that can be given the type $\sigma \rightarrow \tau$ is consistent.

Generator consistency provides a formal characterization of 'abstract classes' in object-oriented languages. An abstract class is one in which some of its components have been left undefined, yet they may be referenced from within the class definition. Since any instance of such a class might invoke an undefined method, a general convention is adopted that instances should not be created for abstract classes.

The consistency condition provides a formal argument that abstract classes must not have instances. Since abstract classes represent inconsistent generators, and the fixpoint is not defined on generator that are not consistent, no instances can be created of an abstract class. This is because instantiation relies upon the fixpoint operator to provide the behavior of the instance.

It is significant that an inconsistent generator in most languages is classified as an error: if a recursive function on integers calls itself with a value that is in the union type of integer or boolean, then the function is simply an error. Although such a function definition does not define a valid function, it could be inherited in such a way that additional functionality is added that completes the definition. Thus inconsistent generators should be thought of as incomplete or partial descriptions of recursive behavior, that can be used as a template that can be completed later in different ways.

4.3 Typing for Inheritance

When applied to inheritance, type theory defines conditions on the validity of generator derivation. Since the self-reference of a derived generator is passed to the generators it inherits, type-constraints are propagated from parents to children.

According to the definition of inheritance, when a generator C inherits from a generator P, the formal argument of C is 'distributed' to P. This means that C must have the form:

$$C = \lambda \mathsf{self}. \cdots P(\mathsf{self}) \cdots$$

If P has been determined to have type $\sigma \rightarrow \tau$, then it follows immediately that the argument type of C must be a subtype of σ. This is the basic constraint induced by inheritance:

> The type of self-reference of an inheritor must be a subtype of the type of self-reference of its parents.

If the child generator, in addition, is to be consistent then its type $C : \sigma' \rightarrow \tau'$ must satisfy σ' subtype τ'. Then, by transitivity, for a child to be consistent its external interface τ' must be a subtype of its parent's self-reference type σ. This is the basic, minimal constraint imposed by inheritance. Note that it is not necessary for the child external interface to be a subtype of the parent external interface.

When applied to function inheritance, the type constraints on inheritance demonstrate that functions can be generalized. A recursive function f of type $\sigma \rightarrow \tau$ can be extended using inheritance to a function f' of type $\sigma' \rightarrow \tau$ as long as $\sigma \leq \sigma'$, which indicates that the new type σ' is more general than σ. The generator G_f of f, which satisfies $f = \text{fix}(G_f)$, has type $(\sigma \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau)$. A wrapper W of f has type

$$W : (\sigma' \rightarrow \tau) \rightarrow (\sigma \rightarrow \tau) \rightarrow (\sigma' \rightarrow \tau)$$

And the inheritance $f' = \text{fix}(W \boxdot G_f)$ denotes a valid function only when $\sigma \leq \sigma'$.

4.4 Encapsulation

Inheritance seems to be a breach of traditional encapsulation and security because references to self cause multiple exits from the syntactic definition of the parent. The effect is that an inheritor is dependent upon the *implementation* of its parent, not just on the parent interface. If the pattern of references the parent makes to itself is changed, the inheritor will be able to detect this change. The traditional notion that data abstraction allows for substitution by behaviorally compatible implementations must be modified in the case of inheritance: recursive behavior must be included in our notion of interface contracts.

Inheritance imposes a responsibility on implementors of a class to use its abstract interface properly. If an operation is provided externally to perform some action, then the methods of the class must call that operation whenever they need achieve the effects of the action. Breaking this rule by 'optimizing' an operation —doing it on the sly without calling the correct abstract operator— is disastrous when combined with inheritance because operations done on the sly cannot be specialized.

Requiring that a record must use its own recursive interfaces properly has an effect on instance variable encapsulation. The standard technique for encapsulating instance variables involves defining a pair of access/assign methods. Inside class methods, however, the variables are usually accessed and assigned directly; the abstract variable interface is bypassed for 'efficiency'. Bypassing the abstract interface prevents variable access from being abstract like other attribute access. However, it is not sufficient for variables to be accessed directly from the parent, as suggested in [25]. Bypassing the virtual interface by a direct access to the parent prevents variable access from being specialized like other virtual attributes. Instance variables shouldn't appear in an abstract interface, but if they do, then only virtual access functions should be invoked.

In defining a wrapper, a choice must be made whether to access an attribute from the parent, from self, or locally. In the definition of a wrapper attribute x, the parent x component may be used to achieve the previous functionality of attribute x. The definition may perform recursion by accessing the virtual x component. Access to other components besides y should always be through the virtual y component, in order to allow for proper use of redefined values. If the attribute y is not redefined, it will access the parent y component anyway.

Local access, which performs the effect of a z operation but does not use the virtual z component, should only be used when the operation is not properly viewed as an abstract use of the z operation. It is almost impossible to justify using an operation in any way but its abstract form. Choosing whether to access a virtual, parent, or local attribute should not be confounded with the virtual/non-virtual decision, which makes the attribute itself virtual or local.

Selective inheritance is a good form of inheritance because it enforces proper use of abstract and virtual interfaces.

Chapter 5

Correctness of the Model

This chapter demonstrates that the inheritance model developed in Chapter 2 characterizes inheritance as used in object-oriented programming languages: a semantics of message sending based on the generator combination is devised and proven equivalent to the traditional operational semantics based on method lookup. The task is simplified by formulating the proof using *method systems*, an abstraction of the state of an object-oriented program consisting of only those aspects relevant to message sending. Other significant aspects of object-oriented languages are abstracted away, including instance variables, assignment, and object creation. This content of this chapter benefited from work by Jens Palsberg Jorgensen, who helped in developing a rigorous proof of the correctness theorem.

5.1 Method Systems

Method systems are a simple formalization of object-oriented programming which support semantics based upon both the denotational and the operational models of inheritance. Method systems encompass only those aspects of object-oriented programming that are directly related to inheritance, or method determination. As such, many important aspects are omitted, including instance variables, assignment, and object creation.

A method system may be understood as part of a snap-shot of an object-oriented system. It consists of all the objects and relationships which exist at a given point during execution of an object-oriented program. The basic ontology for method systems includes instances, classes, and method descriptions, which are mappings from message keys to method expressions. Each object is an instance of a class. Classes have an associated

Method System Domains

Instances	i	$\in$	**Inst**		
Classes	c	$\in$	**Class**		
Message Keys	m	$\in$	**Key**		
Primitives	f	$\in$	**Prim**		
Methods	e	$\in$	**Exp**	:=	self | super | arg
					| $e_1\ m\ e_2$ | $f(e_1,\ \ldots,\ e_q)$

Method System Operations

Class of an instance	*class*	:	**Inst→Class**
Superclass of a class	*parent*	:	**Class→(Class + ?)**
Methods of a class	*methods*	:	**Class→Key→(Exp + ?)**

Table 5.1: Method system domains and their interconnections.

method description and may inherit methods from other classes. These (flat) domains and their interconnections are defined in Figure 5.1. An illustration of a method system is provided in Figure 5.1.

The syntax of method expressions is defined by the **Exp** domain. This domain defines a restricted language used to implement the behavior of objects. For simplicity, methods all have exactly one argument, which is referenced by the symbol arg within the body of the method. Self-reference is denoted by the symbol self, which may be returned as the value of a method, passed as an actual argument, or sent additional messages. A subclass method may invoke the previous definition of a redefined method with the expression super. Message-passing is represented by the expression $e_1\ m\ e_2$, in which the message consisting of the key m and the argument e_2 is sent to the object e_1. Finally, primitive values and computations are represented by the expression $f(e_1,\ \ldots,\ e_q)$. If $q = 0$, then the primitive represents a constant.

class gives the class of an instance. Every instance has exactly one class, although a class may have many instances.

parent defines the inheritance hierarchy which is required to be a tree. For any class c, the value of $parent(c)$ is the parent class of c, or else $\bot_?$ if c is the root. ? is a 1-point domain, consisting of only $\bot_?$. The use of (**Class** + ?) allows us to test monotonically whether a class is the root. Note that + denotes "separated" sum, so that the elements of (**Class** + ?) are (distinguished copies of) the elements of **Class**, the element $\bot_?$, and a new bottom element. All the injections into sum domains are omitted; the meaning of

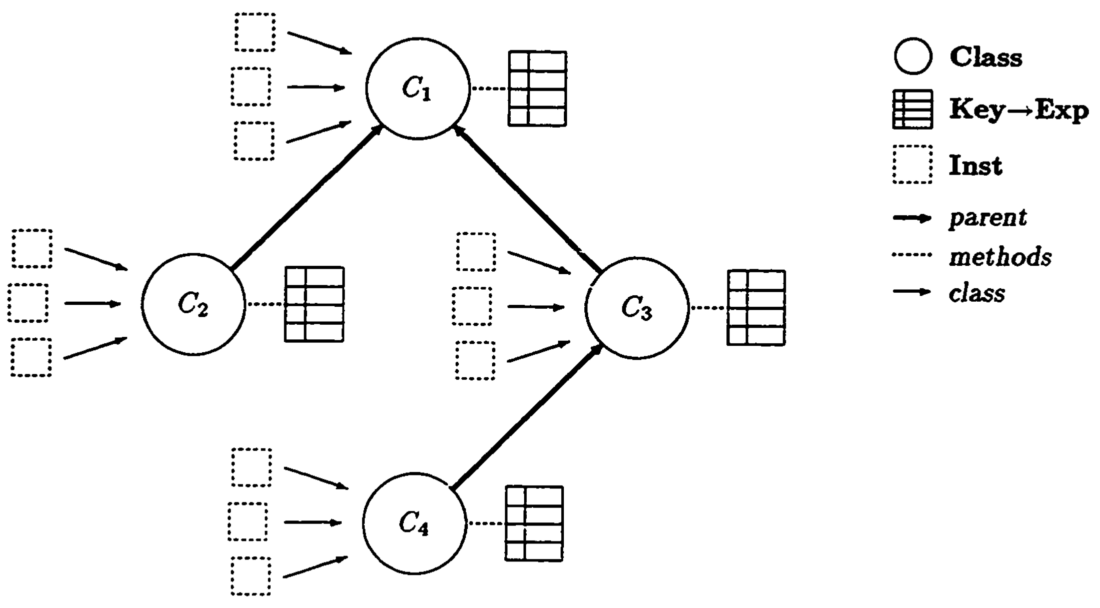


Figure 5.1: A method system.

expressions, in particular $\perp_?$, will always be unambiguously implied by the context.

methods specifies the local method expressions defined by a class. For any class c and any message key m, the value of $(methods\ c\ m)$ is either an expression or $\perp_?$ if c doesn't define an expression for m. Let us assume that the root of the inheritance hierarchy doesn't define any methods. Note that inheritance allows instances of a class to respond to more than the locally defined methods.

In the following two sections the method system is given both a conventional method lookup semantics and a denotational semantics. Both define the result of sending a message to an instance.

5.2 Method Lookup Semantics

The method lookup semantics given in Figure 5.2 closely resembles the implementation of method lookup in object-oriented languages like Smalltalk[11]. It is given in a denotational style due to the abstract nature of method systems. A more traditional operational semantics is not needed because of the absence of updatable storage.

The domains used to represent the behavior an of instance are defined in Figure 5.2. A behavior is a mapping from message keys to *functions* or $\perp_?$. This is clearly contrasted with the methods of a class, which are given by a mapping from message keys to *expressions* or $\perp_?$. Thus a behavior is a semantic entity, while methods are syntactic. Another difference between the behavior of an instance and its class' methods is that the behavior contains a function for every message the class handles, while methods associate an expression only with messages that are different from the class' parent. In the rest of this paper, $\perp$ (without subscript) denotes the bottom element of **Beh**.

The semantics also uses an auxiliary function *root* which determines whether a class is the root of the inheritance hierarchy, see Figure 5.2. **Bool** is the flat 3-point domain of truth values. $[f, g]$ denotes the case analysis of two functions f defined on D_f and g defined on D_g, mapping $x \in D_f + D_g$ to $f(x)$ if $x \in D_f$ or to $g(x)$ if $x \in D_g$.

$$\begin{array}{rcll} & & \mathbf{Num} & \\ a & \in & \mathbf{Val} & = \mathbf{Beh} + \mathbf{Num} \\ s, p & \in & \mathbf{Beh} & = \mathbf{Key} \to ((\mathbf{Val} \to \mathbf{Val}) + ?) \end{array}$$

$$\boxed{\begin{array}{l} \mathit{root} : \mathbf{Class} \to \mathbf{Bool} \\ \mathit{root}(c) = [\lambda\, c_p \in \mathbf{Class}\,.\, \mathit{false}, \lambda\, v \in ?\,.\, \mathit{true}](\mathit{parent}\ c) \end{array}}$$

Table 5.2: Semantics Domains and an auxiliary function.

Sending a message m to an instance i is performed by looking up the message in the instance's class. The lookup process results in a function that takes a message key and an actual argument and computes the value of the message send.

Performing message m in a class c on behalf of an instance i involves searching the sequence of class parents until a method is found to handle the message. This method is then evaluated. In *lookup*, the instance and message remain constant, while the class argument is recursively bound to each of the parents in sequence. At each stage there are two possibilities: (1) the message key has an associated method expression in class c, in which case it is evaluated, and (2) the method is not defined, in which case a recursive call is made to *lookup* after computing the parent of the class. The tail-recursion in *lookup* would be replaced by iteration in a real interpreter.

$$
\begin{array}{l}
\mathit{send} : \mathbf{Inst} \rightarrow \mathbf{Beh} \\
\mathit{send}(i) = \mathit{lookup}(\mathit{class}\ i)\ i \\
\\
\mathit{lookup} : \mathbf{Class} \rightarrow \mathbf{Inst} \rightarrow \mathbf{Beh} \\
\mathit{lookup}(c)\ i = \lambda\, m \in \mathbf{Key}\,.\,[\lambda\, e \in \mathbf{Exp}\,.\,\mathit{do}(e)\ i\ c, \\
\qquad\qquad\qquad\qquad\quad \lambda\, v \in ?\,.\,\mathbf{if}\ \mathit{root}(c) \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{then}\ \bot_? \\
\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ \mathit{lookup}(\mathit{parent}\ c)\ i\ m \\
\qquad\qquad\qquad\qquad\quad](\mathit{methods}\ c\ m) \\
\\
\mathit{do} : \mathbf{Exp} \rightarrow \mathbf{Inst} \rightarrow \mathbf{Class} \rightarrow \mathbf{Val} \rightarrow \mathbf{Val} \\
\mathit{do}([\![\mathsf{self}]\!])\ i\ c\ a = \mathit{send}(i) \\
\mathit{do}([\![\mathsf{super}]\!])\ i\ c\ a = \mathit{lookup}(\mathit{parent}\ c)\ i \\
\mathit{do}([\![\mathsf{arg}]\!])\ i\ c\ a = a \\
\mathit{do}([\![e_1\ m\ e_2]\!])\ i\ c\ a = \mathit{do}(e_1)\ i\ c\ a\ m\ (\mathit{do}(e_2)\ i\ c\ a) \\
\mathit{do}([\![f(e_1,\ \ldots,\ e_q)]\!])\ i\ c\ a = f(\mathit{do}(e_1)\ i\ c\ a,\ \ldots,\ \mathit{do}(e_q)\ i\ c\ a)
\end{array}
$$

$$\begin{tikzcd}
\mathit{send} \arrow[r] & \mathit{lookup} \arrow[r] \arrow[loop above] & \mathit{do} \arrow[loop above] \arrow[ll, bend left] \arrow[l, bend left]
\end{tikzcd}$$

Figure 5.2: The method lookup semantics and its call graph.

Evaluation of methods is complicated by the necessity to interpret occurrences of self and super. The *do* function has three extra arguments, besides the expression being evaluated: the original instance i which received the message whose method is being evaluated, the class c in which the method was found, and an actual argument a. The expression self evaluates to the behavior of the original instance. The expression super requires a continuation of the method search starting from the superclass of the class in which the method occurs. The expression arg evaluates to a. The expression $e_1\ m\ e_2$ evaluates to the result of applying the behavior of the object denoted by e_1 to m and the meaning of the argument e_2.

One important aspect of the method lookup semantics is that it is not "local", in the following sense: the system of functions is essentially mutually recursive, because *do* contains calls to *send* and *lookup*.

5.3 Denotational Semantics

The denotational semantics based on generator modification given in Figure 5.3 uses two additional domains representing behavior generators and wrappers, defined in Figure 5.3. The definition of $\oplus$ is also given in Figure 5.3, using case analysis.

$$\begin{array}{lcl} \mathbf{Gen} & = & \mathbf{Beh}\to\mathbf{Beh} \\ \mathbf{Wrap} & = & \mathbf{Beh}\to\mathbf{Beh}\to\mathbf{Beh} \end{array}$$

$\oplus : (\mathbf{Beh} \times \mathbf{Beh})\to\mathbf{Beh}$

$r_1 \oplus r_2 = \lambda\, m{\in}\mathbf{Key}\,.\,[\lambda\, f{\in}\mathbf{Val}\to\mathbf{Val}\,.\,r_1(m),\ \lambda\, v{\in}?\,.\,r_2(m)](r_1(m))$

Table 5.3: Semantic Domains and $\oplus$.

$\mathit{behave} : \mathbf{Inst}\to\mathbf{Beh}$
$\mathit{behave}(i) = \mathrm{fix}(\mathit{gen}(\mathit{class}\ i))$

$\mathit{gen} : \mathbf{Class}\to\mathbf{Gen}$
$\mathit{gen}(c) = \mathbf{if}\ \mathit{root}(c)$
$\quad\quad \mathbf{then}\ \lambda\, s{\in}\mathbf{Beh}\,.\,\lambda\, m{\in}\mathbf{Key}\,.\,\bot_?$
$\quad\quad \mathbf{else}\ \mathit{wrap}(c)\ \boxed{\triangleright}\ \mathit{gen}(\mathit{parent}\ c)$

$\mathit{wrap} : \mathbf{Class}\to\mathbf{Wrap}$
$\mathit{wrap}(c)\ s\ p = \lambda\, m{\in}\mathbf{Key}\,.\,[\lambda\, e{\in}\mathbf{Exp}\,.\,\mathit{eval}(e)\ s\ p,\ \lambda\, v{\in}?\,.\,\bot_?](\mathit{methods}\ c\ m)$

$\mathit{eval} : \mathbf{Exp}\to\mathbf{Beh}\to\mathbf{Beh}\to\mathbf{Val}\to\mathbf{Val}$
$\mathit{eval}([\![\mathsf{self}]\!])\ s\ p\ a = s$
$\mathit{eval}([\![\mathsf{super}]\!])\ s\ p\ a = p$
$\mathit{eval}([\![\mathsf{arg}]\!])\ s\ p\ a = a$
$\mathit{eval}([\![e_1\ m\ e_2]\!])\ s\ p\ a = \mathit{eval}(e_1)\ s\ p\ a\ m\ (\mathit{eval}(e_2)\ s\ p\ a)$
$\mathit{eval}([\![f(e_1,\ \ldots,\ e_q)]\!])\ s\ p\ a = f(\mathit{eval}(e_1)\ s\ p\ a,\ \ldots,\ \mathit{eval}(e_q)\ s\ p\ a)$

$$\begin{tikzcd} \mathit{behave} \arrow[r] & \mathit{gen} \arrow[loop above] \arrow[r] & \mathit{wrap} \arrow[r] & \mathit{eval} \arrow[loop above] \end{tikzcd}$$

Figure 5.3: The denotational semantics and its call graph.

The behavior of an instance is defined as the fixed-point of the generator of its class. The generator specifies a self-referential behavior, and its fixed-point is that behavior. The generator of the root class produces a behavior in which all messages are undefined.

The generator of a class which isn't the root is created by modifying the generator of the class' parent. The modifications to be made are found in the wrapper of the class, which is a semantic entity derived from the block of syntactic method expressions defined by the class. These modifications are effected by the inheritance function $\boxed{\triangleright}$,

The function *wrap* computes the wrapper of a class as a mapping from messages to the evaluation of the corresponding method, or to $\bot_?$. A wrapper has two behavioral arguments, one used for self-reference, and the other for reference to the parent behavior (i.e. the behavior being 'wrapped'). These arguments may be understood as representing the behavior of **self** and the behavior of **super**. In the definitions, the behavior for **self** is named s and the one for **super** is named p.

The evaluation of a method is always performed in the context of a behavior for **self** (represented by s) and **super** (represented by p). The evaluation of the corresponding expressions, **self** and **super**, is therefore simple. The evaluation of the other expressions is essentially the same as in the method lookup semantics.

Note that each of the functions in the denotational semantics are recursive only within themselves: there is no mutual recursion among the functions, except that which is achieved by the explicit fixed-point.

5.4 Equivalence

The method lookup semantics and the denotational semantics are equivalent because they assign the same behavior to an instance. This proposition is captured by theorem 1.

Theorem 1 *send* = *behave*

The proof of the theorem uses an "intermediate semantics", defined in Figure 5.4, that is inspired by the one used by Mosses and Plotkin [21] in their proof of limiting completeness. The semantics uses $n \in$**Natural**, the flat domain of natural numbers.

The intermediate semantics resembles the method lookup semantics but differs in that each of the syntactic domains of instances, classes, and expressions has a whole family of semantic equations, indexed by natural numbers. The intuition behind the definition is that $send'_n\ i$ allows $(n-1)$ occurrences of **self** on its way before it stops and

$$
\begin{array}{|l|}
\hline
\mathit{send}' : \mathbf{Natural} \to \mathbf{Inst} \to \mathbf{Beh} \\
\mathit{send}'_0\ i = \bot \\
\mathit{send}'_n\ i = \mathit{lookup}'_n\ (\mathit{class}\ i)\ i \qquad n > 0 \\
\\
\mathit{lookup}' : \mathbf{Natural} \to \mathbf{Class} \to \mathbf{Inst} \to \mathbf{Beh} \\
\mathit{lookup}'_0\ c\ i = \bot \\
\mathit{lookup}'_n\ c\ i = \lambda\, m \in \mathbf{Key}\,.\,[\lambda\, e \in \mathbf{Exp}\,.\,\mathit{do}'_n\ e\ i\ c, \qquad\qquad n > 0 \\
\qquad\qquad\qquad\qquad \lambda\, v \in ?\,.\,\mathbf{if}\ \mathit{root}(c) \\
\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ \bot_? \\
\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{lookup}'_n\ (\mathit{parent}\ c)\ i\ m \\
\qquad\qquad\qquad\quad](\mathit{methods}\ c\ m) \\
\\
\mathit{do}' : \mathbf{Natural} \to \mathbf{Exp} \to \mathbf{Inst} \to \mathbf{Class} \to \mathbf{Val} \to \mathbf{Val} \\
\mathit{do}'_0\ e\ i\ c\ a = \bot \\
\mathit{do}'_n\ ([\![\mathsf{self}]\!])\ i\ c\ a = \mathit{send}'_{n-1}\ i \qquad n > 0 \\
\mathit{do}'_n\ ([\![\mathsf{super}]\!])\ i\ c\ a = \mathit{lookup}'_n\ (\mathit{parent}\ c)\ i \qquad n > 0 \\
\mathit{do}'_n\ ([\![\mathsf{arg}]\!])\ i\ c\ a = a \qquad n > 0 \\
\mathit{do}'_n\ ([\![e_1\ m\ e_2]\!])\ i\ c\ a = \mathit{do}'_n\ e_1\ i\ c\ a\ m\ (\mathit{do}'_n\ e_2\ i\ c\ a) \qquad n > 0 \\
\mathit{do}'_n\ ([\![f(e_1,\ \ldots,\ e_q)]\!])\ i\ c\ a = f(\mathit{do}'_n\ e_1\ i\ c\ a,\ \ldots,\ \mathit{do}'_n\ e_q\ i\ c\ a) \qquad n > 0 \\
\hline
\end{array}
$$

Figure 5.4: The intermediate semantics.

gives $\bot$. $\mathit{send}'_n\ i$ is defined in terms of $\mathit{send}'_{n-1}\ i$ via lookup'_n and do'_n because the self expression evaluates to the result of $\mathit{send}'_{n-1}\ i$ which allows one less occurrence of self. (Noticing that the values of $\mathit{lookup}'_0\ c\ i$ and $\mathit{do}'_0\ e\ i\ c\ a$ are irrelevant; let them be $\bot$.)

The following four lemmas state useful properties of the intermediate semantics.

Lemma 1 $\mathit{do}'_n\ e\ i\ c\ a = \mathit{eval}(e)\ (\mathit{send}'_{n-1}\ i)\ (\mathit{lookup}'_n\ (\mathit{parent}\ c)\ i)\ a \qquad n > 0$

Proof: By induction in the structure of e, using the definitions of do' and eval. The base case is proved as follows.

$$\mathit{do}'_n\ ([\![\mathsf{self}]\!])\ i\ c\ a = \mathit{send}'_{n-1}\ i = \mathit{eval}([\![\mathsf{self}]\!])\ (\mathit{send}'_{n-1}\ i)\ (\mathit{lookup}'_n\ (\mathit{parent}\ c)\ i)\ a$$

$$\mathit{do}'_n\ ([\![\mathsf{super}]\!])\ i\ c\ a = \mathit{lookup}'_n\ (\mathit{parent}\ c)\ i = \mathit{eval}([\![\mathsf{super}]\!])\ (\mathit{send}'_{n-1}\ i)\ (\mathit{lookup}'_n\ (\mathit{parent}\ c)\ i)\ a$$

$$\mathit{do}'_n\ ([\![\mathsf{arg}]\!])\ i\ c\ a = a = \mathit{eval}([\![\mathsf{arg}]\!])\ (\mathit{send}'_{n-1}\ i)\ (\mathit{lookup}'_n\ (\mathit{parent}\ c)\ i)\ a$$

The induction step is proven as follows.

$$\mathit{do}'_n\ ([\![e_1\ m\ e_2]\!])\ i\ c\ a$$

$$
\begin{aligned}
&= do'_n\, e_1\; i\; c\; a\; m\, (do'_n\, e_2\; i\; c\; a) \\
&= eval(e_1)\,(send'_{n-1}\; i)\,(lookup'_n\,(parent\; c)\; i)\; a\; m \\
&\qquad (eval(e_2)\,(send'_{n-1}\; i)\,(lookup'_n\,(parent\; c)\; i)\; a) \\
&= eval([\![e_1\; m\; e_2]\!])\,(send'_{n-1}\; i)\,(lookup'_n\,(parent\; c)\; i)\; a
\end{aligned}
$$

$$
\begin{aligned}
&do'_n\,([\![f(e_1, \ldots, e_q)]\!])\; i\; c\; a \\
&\quad= f(do'_n\, e_1\; i\; c\; a, \ldots, do'_n\, e_q\; i\; c\; a) \\
&\quad= f(eval(e_1)\,(send'_{n-1}\; i)\,(lookup'_n\,(parent\; c)\; i)\; a \\
&\qquad\quad , \ldots, eval(e_q)\,(send'_{n-1}\; i)\,(lookup'_n\,(parent\; c)\; i)\; a) \\
&\quad= eval([\![f(e_1, \ldots, e_q)]\!])\,(send'_{n-1}\; i)\,(lookup'_n\,(parent\; c)\; i)\; a
\end{aligned}
$$

QED

Lemma 2 $lookup'_n\; c\; i = gen(c)\,(send'_{n-1}\; i) \qquad n > 0$

Proof: By induction in the number of ancestors of c, using the definitions of *gen*, $\boxed{\triangleright}$, $\oplus$, and *wrap*, lemma 1, and the definition of *lookup'*. In the base case, where c is the root, both sides evaluate to $(\lambda\, m{\in}\mathbf{Key}\,.\,\bot_?)$ because c doesn't define any methods. Then assume that the lemma holds for *parent*(c). The following proof of the induction step uses the definition of *gen* (c isn't the root), the definition of $\boxed{\triangleright}$, the induction hypothesis, the definitions of $\oplus$ and *wrap*, the properties of case analysis, lemma 1, and the definition of *lookup'* (c isn't the root).

$$
\begin{aligned}
&gen(c)\,(send'_{n-1}\; i) \\
&\quad= (wrap(c)\; \boxed{\triangleright}\; gen(parent\; c))\,(send'_{n-1}\; i) \\
&\quad= (wrap(c)\,(send'_{n-1}\; i)\,(gen(parent\; c)\,(send'_{n-1}\; i))) \oplus (gen(parent\; c)\,(send'_{n-1}\; i)) \\
&\quad= (wrap(c)\,(send'_{n-1}\; i)\,(lookup'_n\,(parent\; c)\; i)) \oplus (lookup'_n\,(parent\; c)\; i) \\
&\quad= \lambda\, m{\in}\mathbf{Key}\,.\,[\lambda\, f{\in}\mathbf{Val}{\to}\mathbf{Val}\,.\, f \\
&\qquad\qquad\qquad \lambda\, v{\in}?\,.\, lookup'_n\,(parent\; c)\; i\; m \\
&\qquad\qquad\qquad](wrap(c)\,(send'_{n-1}\; i)\,(lookup'_n\,(parent\; c)\; i)\; m) \\
&\quad= \lambda\, m{\in}\mathbf{Key}\,.\,[\lambda\, f{\in}\mathbf{Val}{\to}\mathbf{Val}\,.\, f \\
&\qquad\qquad\qquad \lambda\, v{\in}?\,.\, lookup'_n\,(parent\; c)\; i\; m \\
&\qquad\qquad\qquad]([\lambda\, e{\in}\mathbf{Exp}\,.\, eval(e)\,(send'_{n-1}\; i)\,(lookup'_n\,(parent\; c)\; i), \\
&\qquad\qquad\qquad\quad \lambda\, v{\in}?\,.\,\bot_? \\
&\qquad\qquad\qquad\quad](methods\; c\; m))
\end{aligned}
$$

$$\begin{aligned}
&= \lambda m{\in}\mathbf{Key}\,.\,[\lambda e{\in}\mathbf{Exp}\,.\,\mathit{eval}(e)\,(\mathit{send}'_{n-1}\ i)\,(\mathit{lookup}'_n\ (\mathit{parent}\ c)\ i),\\
&\qquad\qquad \lambda v{\in}?\,.\,\mathit{lookup}'_n\ (\mathit{parent}\ c)\ i\ m\\
&\qquad\qquad](\mathit{methods}\ c\ m)\\
&= \lambda m{\in}\mathbf{Key}\,.\,[\lambda e{\in}\mathbf{Exp}\,.\,\mathit{do}'_n\ e\ i\ c,\\
&\qquad\qquad \lambda v{\in}?\,.\,\mathit{lookup}'_n\ (\mathit{parent}\ c)\ i\ m\\
&\qquad\qquad](\mathit{methods}\ c\ m)\\
&= \mathit{lookup}'_n\ c\ i
\end{aligned}$$

QED

Lemma 3 $\mathit{send}'_n\ i = (\mathit{gen}(\mathit{class}\ i))^n(\perp)$

Proof: By induction in n, using lemma 2 and the definition of *send′*. In the base case, where $n = 0$, both sides evaluate to $\perp$. Then assume that the lemma holds for $(n-1)$, where $n > 0$. The following proof of the induction step uses the associativity of function composition, the induction hypothesis, lemma 2, and the definition of *send′*.

$$\begin{aligned}
&(\mathit{gen}(\mathit{class}\ i))^n(\perp)\\
&\quad = \mathit{gen}(\mathit{class}\ i)\,((\mathit{gen}(\mathit{class}\ i))^{n-1}(\perp))\\
&\quad = \mathit{gen}(\mathit{class}\ i)\,(\mathit{send}'_{n-1}\ i)\\
&\quad = \mathit{lookup}'_n\ (\mathit{class}\ i)\ i\\
&\quad = \mathit{send}'_n\ i
\end{aligned}$$

QED

Lemma 4 *send′*, *lookup′*, and *do′* are monotone functions of the natural numbers with the usual ordering.

Proof: From lemma 3 it follows that *send′* is monotone. If $n \le m$, then $\mathit{lookup}'_n\ c\ i = \mathit{gen}(c)\ (\mathit{send}'_{n-1}\ i) \sqsubseteq \mathit{gen}(c)\ (\mathit{send}'_{m-1}\ i) = \mathit{lookup}'_m\ c\ i$ using lemma 2, the monotonicity of *send′*, and lemma 2 again. Finally, *do′* is monotone by lemma 1, the monotonicity of *send′* and *lookup′* and lemma 1 again.
QED

Lemma 4 expresses that the family of send'_n 's is an increasing sequence of functions.

$\mathit{interpret} : \mathbf{Inst}{\rightarrow}\mathbf{Beh}$
$\mathit{interpret} = \bigsqcup_n(\mathit{send}'_n\ )$

The following three propositions express the relations among the method lookup semantics, the intermediate semantics, and the denotational semantics.

Proposition 1 *interpret = behave*

Proof:

$$\begin{aligned} interpret(i) &= \bigsqcup_n (send'_n\ i) \\ &= \bigsqcup_n (gen(class\ i))^n(\perp) \\ &= \text{fix}(gen(class\ i)) \\ &= behave(i) \end{aligned}$$

By the definition of *interpret*, lemma 3, the fixed-point theorem, and the definition of *behave*.
QED

Proposition 2 *send* $\sqsupseteq$ *behave*

Proof: The following facts have proofs that are analogous to those of lemma 1 and lemma 2 (the proofs are omitted).

1. $do(e)\ i\ c\ a = eval(e)\ (send(i))\ (lookup(parent\ c)\ i)\ a$
2. $lookup(c)\ i = gen(c)\ (send(i))$

From the definition of *send* and the second fact,

$send(i) = lookup(class\ i)\ i = gen(class\ i)\ (send(i))$.

Hence $send(i)$ is a fixed-point of $gen(class\ i)$. The definition of *behave* expresses that $behave(i)$ is the least fixed-point of $gen(class\ i)$, thus $send(i) \sqsupseteq behave(i)$.
QED

Proposition 3 *send* $\sqsubseteq$ *interpret*

Proof: The functions defined in the method lookup semantics are mutually recursive. Their meaning is the least fixed-point of the generator g which is defined in the obvious way as outlined below.

$$\begin{aligned} D = \ & (\mathbf{Inst} \rightarrow \mathbf{Beh}) \\ & \times (\mathbf{Class} \rightarrow \mathbf{Inst} \rightarrow \mathbf{Beh}) \\ & \times (\mathbf{Exp} \rightarrow \mathbf{Inst} \rightarrow \mathbf{Class} \rightarrow \mathbf{Val} \rightarrow \mathbf{Val}) \end{aligned}$$

Define the generator $g : D \rightarrow D$ for the functions *send*, *lookup*, and *do*:

$$g(s, l, d) = (\lambda\, i \in \mathbf{Inst}\, .\, l(class\ i)\ i, \ldots, \ldots)$$

Now it is proven by induction in n that

$$g^n(\perp_D) \sqsubseteq (send'_n, lookup'_n, do'_n)$$

In the base case, where $n = 0$, the inequality trivially holds. Then assume that the inequality holds for $(n-1)$, where $n > 0$. The following proof of the induction step uses the associativity of function composition, the induction hypothesis, and lemma 4.

$$g^n(\perp_D) = g(g^{n-1}(\perp_D)) \sqsubseteq g(send'_{n-1}, lookup'_{n-1}, do'_{n-1}) \sqsubseteq (send'_n, lookup'_n, do'_n)$$

Now

$$(send, lookup, do) = \text{fix}(g) = \bigsqcup_n g^n(\perp_D) \sqsubseteq \bigsqcup_n (send'_n, lookup'_n, do'_n)$$

and in particular

$$send \sqsubseteq \bigsqcup_n (send'_n) = interpret$$

QED

Proof of theorem 1: Combine propositions 1–3.
QED

This demonstrates that the operational semantics based on method-lookup and the denotational semantics based on generator combination are simply different representations of the same function. Equivalence of the operational and denotational semantics is evidence that the denotational definition is valid. At least for this system, the denotational definition does not seem to be much simpler; it may even be argued that it is a great deal more complex, because it requires an understanding of fixed-points. One advantage of the denotational definition is its direct connection to the standard semantics of programming languages, via fixed-point analysis.

Chapter 6

Denotational Semantics with Generators

This Chapter illustrates how class inheritance can be analyzed within the framework of denotational semantics. A simple language is defined, similar to Gordon's TINY [12], but with classes and inheritance, and its denotational semantics is presented. The significance of this Chapter is that inheritance can be explained using the standard techniques of denotational semantics.

The essential change from previous work on the semantics of classes [28] [13] is the introduction of a domain of class generators as the denotations of class definitions. This domain allows inheritance to be defined as a transformation on class generators.

The shift to explicit generators is significant, for previous attempts to use standard semantics without explicit generators to analyze inheritance in object-oriented languages have required the use of 'syntactic valuations' [31], non-compositional denotations [14], or significant restrictions on the language [22].

6.1 Abstract Syntax

The abstract syntax of the language is given in Table 6.1. Declarations are assumed to be mutually recursive. However, in the variable declaration var I= E, which creates a new storage cell containing the value of E the expression cannot refer to other identifiers defined in the recursive scope of I. Tennent[28] and Hoare[13] discuss ways of avoiding this problem.

Identifiers: $\mathsf{I} \in$ **Ide**

Basic constants: $\mathsf{B} \in$ **Bas**

Binary operators: $\mathsf{O} \in$ **Opr**

Programs: $\mathsf{P} \in$ **Prog** ::= E

Declarations: $\mathsf{D} \in$ **Dcl** ::=

V	variables
M	procedures/classes
private D_1 in D_2	local declaration
Λ	empty

Variables: $\mathsf{V} \in$ **Var** ::=

var $\mathsf{I} = \mathsf{E}$	variables
$\mathsf{V}_1;\ \mathsf{V}_2$	
Λ	empty

Complex Constants: $\mathsf{M} \in$ **Proc** ::=

proc $\mathsf{I}(\vec{\mathsf{I}})\ \mathsf{E}$	procedures/functions
class $\mathsf{I}\ \mathsf{I}'\ \mathsf{D}$	classes
$\mathsf{D}_1;\ \mathsf{D}_2$	mutual recursive binding
Λ	empty

Expressions: $\mathsf{E} \in$ **Exp** ::=

I	identifiers
B	basic constants
new E	object creation
$\mathsf{E}_1.\mathsf{E}_2$	field/method selection
$\mathsf{E}_1\ \mathsf{O}\ \mathsf{E}_2$	binary operator
$\mathsf{E}(\vec{\mathsf{E}})$	function/procedure application
let D in E	local declaration
$\mathsf{E}_1 := \mathsf{E}_2$	assignment
$\mathsf{E}_1;\ \mathsf{E}_2$	sequence
if E_1 then E_2 else E_3	conditional
while E_1 do E_2	iteration

Table 6.1: Abstract syntax of simple inheritance language.

The class declaration is derived directly from Smalltalk. The declaration class $\mathsf{I}\ \mathsf{I}'\ \mathsf{D}$ defines a class named I that inherits from its parent class named I'. A predefined class Base is used when no other parent is desired. Each instance of the class is an instantiation of the declaration D. This is typically a declaration of the form private D_1 in D_2, where

D_1 represents the hidden local state of each instance (typically a collection of variable declarations), while D_2 represents the external attributes of the instances (typically a collection of procedures). For example, the following is a pair of class definitions in this language:

```
class Counter Base
private
    value
in
    proc increment()
        value := value + 1;
    proc limit(bound)
        while value < bound do
            self.increment();


class StepCounter
inherit Counter
private
    step
    in
        fun increment;
            for i := 1 to step do super.increment;
```

Unlike Smalltalk, methods may only refer to variables declared in the same class definition, not to inherited variables.[1] The *pseudovariables* self and super provide access to the methods of the class and of the superclass respectively. Methods in a class instance are accessed by the expression $E_1.E_2$.

[1] Little extra effort is necessary to allow access to parent variables (see Section 8).

Numbers			**Num**		
Booleans	β	$\in$	**Bool**		
Locations	ι	$\in$	**Loc**		
Answers			**Ans**		
Storable values	μ	$\in$	**Sv**	=	**Bool** + **Num** + **Fun** + **Env** +**Env** + {unbound}
Denotable values	δ	$\in$	**Dv**	=	**Sv** + **Loc** + **Cla** + $\mathbf{Dv}^*$
Stores	σ	$\in$	**Sto**	=	**Loc**→(**Sv** + {unused})
Environments	ρ	$\in$	**Env**	=	**Ide**→(**Dv** + {undefined})
Specialized environments	ρ	$\in$	$\mathbf{Env_D}$	=	**Ide**→(**D** + {undefined})
Environment generators	γ	$\in$	**Gen**	=	**Env**→**Env**
Command continuations	θ	$\in$	**Cc**	=	**Sto**→**Ans**
Generic continuations	Cont(D)			=	**D**→**Cc**
Expression continuations	κ	$\in$	**Ec**	=	**Cont(Dv)**
Functions	ϕ	$\in$	**Fun**	=	**Ec**→**Ec**
Commands	ϑ	$\in$	**Cmd**	=	**Ec**→**Cc**
Declaration continuations	χ	$\in$	**Dc**	=	**Cont(Gen)**
Classes	ζ	$\in$	**Cla**	=	**Cont(Dc)**
Wrappers			**Wrap**	=	**Env**→**Env**→**Env**

Table 6.2: Semantic domains for inheritance.

6.2 Semantic Domains

The semantic domains for the analysis of the simple language are defined in Table 6.2. A semantic domain may have an associated Greek letter, which acts as a general meta-variable for values of that domain. Subscripts are used when more than one variable in a domain is needed. For domain with meta-variable ν, the empty or null value in the domain is denoted $\nu_\emptyset$. For example, ρ, the generic environment variable **Env**, is used to represent the empty environment $\rho_\emptyset$. The domain definitions form a system of recursive domain equations, whose solution provides appropriate lattice structure for the identification of fixpoints [24].

One of the major premises of standard semantics is that *environments*, which define the static denotation of identifiers, are cleanly differentiated from *stores*, which contain dynamically updatable locations. Using these two concepts, variables is understood as denoting locations, while locations index a value in the store. The domains of *denotable* and *storable* values are also differentiated. Typically the storable values are a subset of the denotable values; in a language without pointers, locations are not storable (though they are denotable). The content of these domains determines the expressive power of the language in question: the domain of denotable values determines what kinds of structures can be bound to identifiers and thus referred to from within the program; while the domain of storable values determines the kinds of structures that may stored and manipulated during the dynamic computation sequence of programs.

The storable domain consists of basic constants, abstractions, and environments (which represent record values). These values may be manipulated dynamically by placing them in the store. The denotable values include all these but have in addition the domain of locations and classes. Locations represent the place in the store where the value of a variable may be found. A variable always denotes a location; the value of the variables is found in the store. *Pointer variables* are not supported, because locations are not storable.

The environment and store, as a mapping from identifiers or locations to values, are treated as records, and updated using the preferential record combination functions defines in Section 2.1.1. This combination function is essentially equivalent to the function *divert* commonly used in denotational semantics. Though somewhat unconventional, the use of $\oplus$ is justified because it provides uniformity, in that the same operations and notation is wherever the environment/store/record concept appears. In addition, it facilitates the introduction of inheritance mechanisms into standard semantics.

The semantic domains also include a number of continuation domains which are used to represent the meaning of pieces of programs. A continuation is a function that represents 'the rest of the program'. A command continuation is passed the current state of the store, on which it performs the rest of the computation of the program. Parameterized continuations require another value to be passed in addition to the store; this value often represents the result of a previous computation, which the continuation needs to resume computation.

In addition to the conventional domains of standard semantics, discussed above, a domain of environment generators is introduced. The explicit domain of generators makes inheritance possible. However, environment generators also provide an elegant solution to the problem of mutual recursion. The traditional declaration continuation accepts a 'little environment', making mutual recursion difficult to specify, because allocation of variables is intermixed with recursive constants. This problem is solved by passing environment generators to declaration continuations instead.

Classes are denoted by continuations that allocate storage for instances, and pass the resulting environment generator of external identifiers to the continuation argument.

6.3 Semantic Clauses

The translation from syntactic to semantic domains is defined by the following semantic clauses. Range checking on domains has been omitted, but is indicated mnemonically by the choice of variables.

Despite the fact that environment generators are used in the semantics, environments are still passed to the valuations as in conventional semantics. These 'static' environments represent the external context of the declaration, while the generator's bound environment variable represents only identifiers in the same mutually recursive scope. Other arrangements are possible.

$\mathcal{P} : \mathbf{Prog} \rightarrow \mathbf{Ans}$

$$\mathcal{P}[\![\mathsf{E}]\!] = \mathbf{clet}\ \delta = \mathcal{E}[\![\mathsf{E}]\!]([\mathsf{Base} \mapsto \lambda\chi\,.\,\chi(\gamma_\emptyset)])\ \mathbf{in}\ \ \textit{print-answer}(\delta)$$

An expression representing a program is evaluated in an initial environment, and provided with a continuation which converts the value of the expression into an answer in **Ans**. The initial environment contains a definition for the class Base, which serves as the starting-point for inheritance. The Base class does not define any variables or methods. The value $\lambda\chi\,.\,\chi(\gamma_\emptyset)$: **Cla** is a function of a subclass continuation, to which the parent attributes are normally passed. Since Base doesn't define any attributes, it passes a null generator.

$\mathcal{D} : \mathbf{Dcl} \rightarrow \mathbf{Env} \rightarrow \mathbf{Dc} \rightarrow \mathbf{Cc}$

$$\mathcal{D}[\![\mathsf{class\ I\ I'\ D}]\!]\rho\chi = \chi(\lambda\rho'\,.\,[\mathsf{I} \mapsto \lambda\chi'.\mathbf{clet}\ \gamma = \rho(\mathsf{I'})\ \mathbf{in}\ \ \mathbf{clet}\ \omega = \mathcal{W}[\![\mathsf{D}]\!](\rho' \oplus \rho)\ \mathbf{in}\ \ \chi'(\omega \boxed{\triangleright} \gamma)])$$

$\mathcal{W} : \mathbf{Dcl} \rightarrow \mathbf{Env} \rightarrow \mathbf{Cont(Wrap)} \rightarrow \mathbf{Cc}$

$$\mathcal{W}[\![\mathsf{D}]\!]\rho\chi = \mathbf{clet}\ \omega = \mathcal{D}[\![\mathsf{D}]\!]\rho\ \mathbf{in}\ \ \chi(\lambda\rho_s\,.\,\lambda\rho_p\,.\,\omega([[\mathsf{self} \mapsto \rho_s, \mathsf{super} \mapsto \rho_p]]))$$

A class declaration takes the class environment generator, combines in the class being defined, and passes the result to the rest of the program. The denotation of the class definition is the composition of the parent class denotation, retrieved by $\rho(\mathsf{I}')$, with the local attributes of the class, allocated by $\mathcal{W}[\![\mathsf{D}]\!]$. Since class denotations are allocators, this has the effect of first allocating the parent, resulting in a method generator γ that is passed to the subclass allocator, which allocates the subclass declaration and combines its methods with the parent method generator.

The methods of the class are converted into a wrapper. The environment ρ_s represents self-reference within the methods, and is bound to self, while ρ_p represents parent methods, and is bound to **super**. These two symbols are available within the method body.

The method wrapper is then combined with the parent generator and passed to the continuation. Inheritance occurs at this point in the methods, because of the distributive nature of wrapper application.

$$\mathcal{D} : \mathbf{Dcl} \rightarrow \mathbf{Env} \rightarrow \mathbf{Dc} \rightarrow \mathbf{Cc}$$

$$\mathcal{D}[\![\mathsf{M}]\!]\rho\chi = \chi(\lambda\rho' . \mathcal{M}[\![\mathsf{M}]\!]((\rho'|\mathcal{B}[\![\mathsf{M}]\!]) \oplus \rho))$$

$$\begin{aligned}
\mathcal{D}[\![\mathsf{V}]\!]\rho\chi &= \textbf{clet } \rho' = \mathcal{V}[\![\mathsf{V}]\!] \textbf{ in} \\
&\qquad \chi(\lambda\rho . \rho') \\
\mathcal{D}[\![\mathsf{D}_1;\ \mathsf{D}_2]\!]\rho\chi &= \textbf{clet } \gamma_1 = \mathcal{D}[\![\mathsf{D}_1]\!]\rho \textbf{ in} \\
&\qquad \textbf{clet } \gamma_2 = \mathcal{D}[\![\mathsf{D}_2]\!]\rho \textbf{ in} \\
&\qquad\quad \chi(\gamma_1 \boxed{\oplus_\perp} \gamma_2) \\
\mathcal{D}[\![\mathsf{private}\ \mathsf{D}_1\ \mathsf{in}\ \mathsf{D}_2]\!]\rho\chi &= \textbf{clet } \gamma = \mathcal{D}[\![\mathsf{D}_1]\!]\rho \textbf{ in} \\
&\qquad \mathcal{D}[\![\mathsf{D}_2]\!](\mathrm{fix}(\gamma) \oplus \rho)\chi \\
\mathcal{D}[\![\Lambda]\!]\rho\chi &= \chi(\gamma_\emptyset)
\end{aligned}$$

The generator representing a mutually recursive declaration is has its self-reference environment limited to only those identifiers bound in the declaration. This prevents a violation of block structure in which an identifier in the outside environment could be rebound during inheritance. The valuation $\mathcal{B}$ returns the set of identifiers bound in a declaration.

Two additional valuations are used to separate the interpretation of variables and allocation from the interpretation of procedures and mutual recursion:

$\mathcal{V}$: **Var→Cont(Env)→Cc**

$$\mathcal{V}[\![\mathsf{var\ I = E}]\!]\rho\kappa = \mathcal{E}[\![\mathsf{E}]\!]\rho(\mathit{store}\{\lambda\iota.\kappa([\mathsf{I} \mapsto \iota])\})$$

$$\begin{aligned}\mathcal{V}[\![\mathsf{V_1;\ V_2}]\!]\kappa = {} & \mathbf{clet}\ \rho_1 = \mathcal{V}[\![\mathsf{V_1}]\!]\ \mathbf{in}\\ & \quad \mathbf{clet}\ \rho_2 = \mathcal{V}[\![\mathsf{V_2}]\!]\ \mathbf{in}\\ & \qquad \kappa(\rho_1 \oplus \rho_2)\end{aligned}$$

$\mathcal{M}$: **Dcl→Gen**

$$\mathcal{M}[\![\mathsf{proc\ I(\vec{I})\ E}]\!] = \lambda\rho.[\mathsf{I} \mapsto \lambda\kappa\vec{\delta}.\mathcal{E}[\![\mathsf{E}]\!]([\vec{\mathsf{I}} \mapsto \vec{\delta}] \oplus \rho)\kappa]$$

$$\mathcal{M}[\![\mathsf{M_1;\ M_2}]\!] = \mathcal{M}[\![\mathsf{M_1}]\!]\ \boxed{\oplus_\perp}\ \mathcal{M}[\![\mathsf{M_2}]\!]$$

Preferential combination $\oplus$ is used for extending an environment with new identifiers that may override existing ones, while strict combination $\oplus_\perp$ is used to combine the 'little environments' in a series declaration to prevent multiple definition of an identifier in a single declaration.

$\mathcal{E} : \mathbf{Exp} \rightarrow \mathbf{Env} \rightarrow \mathbf{Ec} \rightarrow \mathbf{Cc}$

$$
\begin{aligned}
&\mathcal{E}[\![\mathsf{I}]\!]\rho\kappa = \textbf{if } \rho[\![\mathsf{I}]\!] = \mathsf{undefined} \textbf{ then } \textit{error} \textbf{ else } \textit{lookup}\kappa(\rho[\![\mathsf{I}]\!])\\
&\mathcal{E}[\![\mathsf{B}]\!]\rho\kappa = \kappa(\mathcal{B}[\![\mathsf{B}]\!])\\
&\mathcal{E}[\![\mathsf{new\ E}]\!]\rho\kappa = \textbf{clet } \zeta = \mathcal{E}[\![\mathsf{E}]\!]\rho \textbf{ in}\\
&\qquad\qquad \textbf{clet } \gamma = \zeta \textbf{ in}\\
&\qquad\qquad\quad \kappa(\mathrm{fix}(\gamma))\\
&\mathcal{E}[\![\mathsf{E}_1.\mathsf{E}_2]\!]\rho\kappa = \textbf{clet } \rho' = \mathcal{E}[\![\mathsf{E}_1]\!]\rho \textbf{ in}\\
&\qquad\qquad \mathcal{E}[\![\mathsf{E}_2]\!](\rho' \oplus \rho)\kappa\\
&\mathcal{E}[\![\mathsf{E}_1 \mathsf{\ O\ } \mathsf{E}_2]\!]\rho\kappa = \textbf{clet } \delta_1 = \mathcal{E}[\![\mathsf{E}_1]\!]\rho \textbf{ in}\\
&\qquad\qquad \textbf{clet } \delta_2 = \mathcal{E}[\![\mathsf{E}_2]\!]\rho \textbf{ in}\\
&\qquad\qquad\quad \kappa(\mathcal{O}[\![\mathsf{O}]\!]\delta_1\delta_2)\\
&\mathcal{E}[\![\mathsf{E}(\vec{\mathsf{E}})]\!]\rho\kappa = \textbf{clet } \phi = \mathcal{E}[\![\mathsf{E}]\!]\rho \textbf{ in}\\
&\qquad\qquad \textbf{clet } \vec{\delta} = \vec{\mathcal{E}}[\![\vec{\mathsf{E}}]\!]\rho \textbf{ in}\\
&\qquad\qquad\quad \phi\kappa\vec{\delta}\\
&\mathcal{E}[\![\mathsf{let\ D\ in\ E}]\!]\rho\kappa = \textbf{clet } \gamma = \mathcal{D}[\![\mathsf{D}]\!]\rho \textbf{ in}\\
&\qquad\qquad \mathcal{E}[\![\mathsf{E}]\!](\mathrm{fix}(\gamma) \oplus \rho)\kappa\\
&\mathcal{E}[\![\mathsf{E}_1 := \mathsf{E}_2]\!]\rho\kappa = \textbf{clet } \iota = \mathcal{E}[\![\mathsf{E}_1]\!]\rho \textbf{ in}\\
&\qquad\qquad \textbf{clet } \delta = \mathcal{E}[\![\mathsf{E}_2]\!]\rho \textbf{ in}\\
&\qquad\qquad\quad \textit{update}(\kappa\delta)(\iota, \delta)\\
&\mathcal{E}[\![\mathsf{E}_1;\ \mathsf{E}_2]\!]\rho\kappa = \textbf{clet } \delta = \mathcal{E}[\![\mathsf{E}_1]\!]\rho \textbf{ in}\\
&\qquad\qquad \mathcal{E}[\![\mathsf{E}_2]\!]\rho\kappa\\
&\vec{\mathcal{E}}[\![\mathsf{E} \parallel \vec{\mathsf{E}}]\!]\rho\kappa = \textbf{clet } \delta = \mathcal{E}[\![\mathsf{E}]\!]\rho \textbf{ in}\\
&\qquad\qquad \textbf{clet } \vec{\delta} = \vec{\mathcal{E}}[\![\vec{\mathsf{E}}]\!]\rho \textbf{ in}\\
&\qquad\qquad\quad \kappa(\delta \parallel \vec{\delta})\\
&\vec{\mathcal{E}}[\![\langle\rangle]\!]\rho\kappa = \kappa(\langle\rangle)
\end{aligned}
$$

These definitions for the most part are conventional standard semantics. The valuation $\mathcal{O} : \mathbf{Opr} \rightarrow \mathbf{Dv} \rightarrow \mathbf{Dv} \rightarrow \mathbf{Dv}$ which associates functions with primitive operators is left unspecified. The most significant difference is that the fixpoint of a class is taken when instances are created by new, rather than when the class is declared. Thus the fixpoint is associated with instantiation rather than declaration. In addition the let construct must take a fixpoint before combining the local declaration with the environment.

The following auxiliary definitions are the standard functions for manipulating stores.

$error : \mathbf{Cc}$

$lookup : \mathbf{Cont(Dv)} \rightarrow \mathbf{Cont(Loc)}$

$lookup\kappa = \lambda\iota\,.\,\lambda\sigma\,.\,\mathbf{if}\ \sigma(\iota) = \text{unused}\ \mathbf{then}\ error\sigma\ \mathbf{else}\ \kappa(\sigma(\iota))\sigma$

$update : \mathbf{Cc} \rightarrow \mathbf{Cont(Loc \times Sv)}$

$update\theta = \lambda\iota,\mu\,.\,\lambda\sigma\,.\,\theta([\,\iota \mapsto \mu\,] \oplus \sigma)$

$rv : \mathbf{Cont(Sv)} \rightarrow \mathbf{Cont(Dv)}$

$rv\kappa = \lambda\delta\,.\,\mathbf{if}\ \delta \in \mathbf{Loc}\ \mathbf{then}\ lookup\kappa\delta\ \mathbf{else}\ \kappa\delta$

$lv : \mathbf{Cont(Loc)} \rightarrow \mathbf{Cont(Dv)}$

$lv\kappa = \lambda\delta\,.\,\mathbf{if}\ \delta \in \mathbf{Loc}\ \mathbf{then}\ \mathbf{clet}\ \iota = new\ \mathbf{in}\ update(\kappa\iota)(\iota,\delta)\ \mathbf{else}\ \kappa\delta$

$store : \mathbf{Cont(Loc)} \rightarrow \mathbf{Cont(Dv)}$

$store = rv \circ lv$

Chapter 7

Inheritance in Simula

A Simula class consists of a declaration and an imperative. The declaration defines the *attributes* possessed by instances of the class. The imperative is an expression which is evaluated when an instance is created; it provides the primary behavior of the class. Thus a Simula class may be viewed as a combination of the block and procedure concepts.

Simula introduced inheritance, or subclassing, by allowing new classes to be built upon a *prefix* class. A subclass was viewed as textually extending its prefix. The subclass inherits all the attributes of its prefix, while its imperative is invoked from the prefix imperative with the inner statement. Textual concatenation is complicated by allowing the subclass to redefine attributes given its prefix; in addition, attributes may be identified as *virtual* in the prefix, in which case a redefined attributes will be used by the prefix.

Attributes may be accessed from outside the instance by a *reference* to the instance. A reference may be *qualified* by a class name (which must be one of the superclasses of the instance), to allow access to an instance as if it were an instance of one of its superclasses. Thus qualification resembles a form of coercion on references. Qualification is useful because a subclass may *redefine* the procedures or functions of its superclass, and qualification is necessary to access the occluded definition from within a subclass. Redefinition of non-virtual attributes does not affect the behavior of superclasses which use the redefined identifier; superclasses continue to reference the occluded definition. Virtual attributes, however, have only *one* accessible definition for all levels of classes, which is that given by the last redefinition. Thus a superclass will reference the redefined version of a virtual attribute. Virtual attributes are said to be *qualification independent*.

Classes			Class	::=	I' class I virtual $\vec{I}$ D E
Expressions	E	∈	Exp	::=	this | inner

Table 7.1: Simula syntax.

In addition, Simula provides mechanisms for coroutine programming, though these will not be examined here, as they do not have much impact on the inheritance mechanism.

7.1 Syntax

The syntax for discussing Simula is given in Table 7.1. The declaration I' class I virtual $\vec{I}$ D E defines a class named I with prefix class I'. It defines local attributes D and has an imperative E. It declares the attributes $\vec{I}$ to be virtual. The virtual attributes of a class are the virtual attributes of its prefix unioned with the locally declared virtual identifiers; once an identifier is virtual, it is virtual in all subclasses.

The attributes declared by a class, together with all those of its prefixes, may be used in both attribute expressions and the imperative. However, only the last redefinition of any attribute can be accessed directly by name.

The keyword this, a predefined reference variable, is used to access attributes according to the class in which they where defined. Using the name of a prefix class p, the the previous definition of an redefined attribute a with the expression this.p.a. As mentioned above, this cannot yield previous definitions of virtual attributes because they are qualification independent; hence this is not directly analogous to super in Smalltalk.

inner is used in an imperative to invoke a subclass imperative, if there is one. If inner is used when no subclass exists, then it has no effect. Though inner is technically a statement in Simula, it conversion to an expression, for the sake of uniformity, has little effect upon its use.

Class parameters have been omitted; however, their introduction poses few problems, being merely a combination of the procedure and class mechanisms.

A Simula class definition is presented below. The class Counter has three attributes: the integer value, an increment procedure, and a procedure limit that increments the value until it is not less than a bound. Its imperative just initializes the value to zero.

```
class Counter
    integer value;
    virtual increment;
    procedure increment();
        value := value + 1;
    procedure limit(bound);
        while value < bound do
            this.increment();
begin
    value := 0
    inner
end
```

The following class is an extension of Counter which allows increments other than one.

```
Counter class StepCounter
    integer step;
    procedure increment;
        value := value + step;
    begin
        step := 1;
        inner
    end
```

Note that increment cannot be defined to invoke the increment function of the parent class Counter, as in Smalltalk and Flavors. This is because because virtuals are independent of qualification so that this(Counter).increment within class StepCounter would still refer to the redefined increment.

7.2 Domains

Simula requires additional domains for references and imperatives. These domains are defined below:

Class behavior	S, P, C	$\in$	**Beh**	$=$	[
Imperative					*imp* : **Cmd→Cmd**
Attributes					*attr* : **Env**
Attribute definitions					*def* : **Env**
Virtual identifiers					*virt* : $\mathbf{Ide}^*$
Qualified reference					*this* : **Ref**
					]
Class specifications	G	$\in$	**Inst**	$=$	**Beh→Beh**
Class-spec continuations	ψ	$\in$	**InstCont**	$=$	**Cont(Inst)**
Classes	ζ	$\in$	**Cla**	$=$	**Cont(InstCont)**

Table 7.2: Simula domains

Additional Simula Domains

References			**Ref**	$=$	**Ide→Env**
Denotable values	δ	$\in$	**Dv**	$=$	**Sv + Loc + Cla + Cmd**

A reference provides access to the attributes of an instance at each level of its prefixing. To access an attribute, a reference is first qualified by a class name to select the prefix level, and then the attribute name is used to select the desired value. Thus a reference is a mapping from class identifiers to attribute identifiers to values; using the notion of records or environments, it is an environment of environments. Since environments are already included in the domain of denotable values, and environments may contain environment values, the introduction of a reference domain is not strictly necessary.

A command is a partially evaluated expression. It resembles a function but does not require an argument value. Commands are used to represent imperatives, which are stored in the standard environment under the keyword symbol **inner**, so they must be included in the domain of denotable values.

The denotation of a Simula class has several interdependent components. Besides a parameterized imperative and the environment of attributes, it is necessary to include the set of virtual identifiers and a complete reference used to interpret the pseudovariable **this**. Domains for representing these structures are defined formally in Table 7.2.

The basic attributes of a class are given in the domain of class behavior, **Beh**, which expressed as a record type indexed by the special symbols *imp*, *attr*, *def*, *virt* and *this*. The imperative *imp* is a function on commands in order to allow parameterization by a subclass imperative (the formal parameter of the imperative represents **inner**). The attribute environment *attr* contains the denotation of identifiers available for use within the class, which may be changed by redefinition of virtuals. The attribute definition environment, on the other hand, contains the values defined by the class. Clearly *attr* and *def* differ only when virtuals are redefined.

7.3 Semantics

The semantics of inheritance is developed as a system of equations which specify the meaning of a subclass in terms of the parent behavior and syntax of the subclass definition. The parent behavior P has components P_{imp}, P_{def}, P_{virt}, P_{attr} and P_{this}. Self-reference is provided by the variable S, of which only the attributes S_{attr} are used. Using these structures, the child components C_{imp}, C_{def}, C_{virt}, C_{attr} and C_{this} are developed as a system of equations.

The variable γ represents the result of the local class declaration $\mathcal{D}[\![\mathsf{D}]\!]\rho$. Using the notation of continuations, the system of equations is defined in the context of **clet** $\gamma = \mathcal{D}[\![\mathsf{D}]\!]\rho$ **in**.

...

The attributes defined by a class are those declared by the class, evaluated in an environment of class identifiers, combined with the parent's defined attributes:

$$C_{def} = \gamma(\rho') \oplus P_{def}$$

The environment supplied to the subclass declaration contains all the attributes defined in the class along with the binding of the special symbol this:

$$\rho' = [\mathsf{this} \mapsto C_{this}] \oplus C_{attr}$$

The attributes available on a class are the virtual attributes (limited to just those virtuals available at this level) combined with the attributed defined by the class.

$$C_{attr} = S_{attr}|C_{virt} \oplus C_{def}$$

The virtual attributes names of a class are just those declared in the class plus those of the parent.

$$C_{virt} = \vec{\mathsf{I}} \cup P_{virt}$$

this is interpreted adding an association of the class name with the attributes of the class to the interpretation of this in the parent.

$C_{this} = [\mathsf{I} \mapsto C_{\mathbf{attr}}] \oplus P_{this}$

The imperative of the subclass is the composition of the superclass imperative with the imperative part defined in the subclass. This has the effect of binding **inner** in the parent to the subclass imperative, while allowing for the possibility of future binding of any **inner** statements appearing in the subclass.

$C_{imp} = \lambda\vartheta . P_{imp}(\mathcal{E}[\![\mathsf{E}]\!]([\mathsf{inner} \mapsto \vartheta] \oplus \rho'))$

The subclass imperative is bound as a command to **inner** in the environment. The **inner** statement is evaluated by invoking the value from the environment:

$$\boxed{\mathcal{E}[\![\mathsf{inner}]\!]\rho\kappa = \rho(\mathsf{inner})\kappa}$$

A class is instantiated to produce a generator, whose fixpoint is a complete behavioral specification of the instance. From this the imperative is selected and applied to a null command to specify an empty **inner**. The continuation expecting a reference to the new instance is passed the reference component *this*. The resulting command continuation takes control after the imperative is complete.

$$\boxed{\begin{aligned}\mathcal{E}[\![\mathsf{new\ E}]\!]\rho\kappa = \ & \mathbf{clet}\ \zeta = \mathcal{E}[\![\mathsf{E}]\!]\rho\ \mathbf{in} \\ & \quad \mathbf{clet}\ G = \zeta\ \mathbf{in} \\ & \quad\quad \mathbf{let}\ C = \mathrm{fix}(G)\ \mathbf{in} \\ & \quad\quad\quad C_{imp}\theta_{\emptyset}(\kappa(C_{this}))\end{aligned}}$$

The base class **Prefix** represents the null class definition from which all other classes inherit. It has no attributes, all its environments are empty, and provides a null imperative:

$\mathsf{Prefix}_{imp} = \lambda\phi\kappa . \phi(\kappa)$

The complete semantic valuation for Simula class declarations, composed of the parts outlined above, is defined as follows:

$$\begin{array}{l}
\mathcal{A}[\![\mathsf{I'\ class\ I\ virtual\ \vec{I}\ D\ E}]\!]\rho = \\
\quad \textbf{clet}\ \zeta = \mathcal{E}[\![\mathsf{I'}]\!]\rho\ \textbf{in} \\
\quad\quad \textbf{clet}\ G = \zeta\ \textbf{in} \\
\quad\quad\quad \textbf{clet}\ \gamma = \mathcal{D}[\![\mathsf{D}]\!]\rho\ \textbf{in} \\
\quad\quad\quad\quad [\mathsf{I} \mapsto (\lambda S.\lambda P.C) \boxed{\cdot} G] \\
\quad\quad \textbf{where}\ C_{imp} = \lambda\vartheta.P_{imp}(\mathcal{E}[\![\mathsf{E}]\!]([\mathsf{inner} \mapsto \vartheta] \oplus \rho')) \\
\quad\quad\quad\quad\quad C_{attr} = S_{attr}|C_{virt} \oplus C_{def} \\
\quad\quad\quad\quad\quad C_{def} = \gamma(\rho') \oplus P_{def} \\
\quad\quad\quad\quad\quad C_{virt} = \vec{\mathsf{I}} \cup P_{virt} \\
\quad\quad\quad\quad\quad C_{this} = [\mathsf{I} \mapsto C_{attr}] \oplus P_{this} \\
\quad\quad\quad\quad\quad \rho' = [\mathsf{this} \mapsto C_{this}] \oplus C_{attr}
\end{array}$$

The fact that so much information is required to describe Simula classes in such a way that inheritance is fully specified is an indication that classes are not very well encapsulated.

7.4 Example

The translation of the Counter class introduced above is sketched in this section.

$$\begin{array}{l}
C_{imp} = \lambda\vartheta.\lambda\kappa.\mathit{update}(\vartheta\kappa)(\iota_1, 0) \\
C_{this} = [\mathsf{Counter} \mapsto C_{attr}] \\
C_{attr} = S_{attr}|\{\mathsf{increment}\} \oplus C_{def} \\
\rho' = [\mathsf{this} \mapsto C_{this}] \oplus C_{attr} \\
C_{def} = [\ \mathsf{value} \mapsto \iota_1, \\
\quad\quad\quad \mathsf{increment} \mapsto \mathcal{E}[\![\mathsf{value := value + 1}]\!]\rho', \\
\quad\quad\quad \mathsf{limit} \mapsto \lambda\kappa\delta.\mathcal{E}[\![\mathsf{while\ value < bound\ do\ increment}]\!]([\mathsf{bound} \mapsto \delta] \oplus \rho')\kappa] \\
C_{virt} = \{\mathsf{increment}\}
\end{array}$$

The fixpoint of this class construct, which specifies the behavior of instances, is illustrated below:

$$
\begin{aligned}
C_{imp} &= \lambda\vartheta.\lambda\kappa.\,update(\vartheta\kappa)(\iota_1, 0)\\
C_{this} &= [\text{Counter} \mapsto C_{attr}]\\
C_{attr} &= [\ \text{value} \mapsto \iota_1,\\
&\qquad \text{increment} \mapsto \mathcal{E}[\![\text{value} := \text{value} + 1]\!]\rho',\\
&\qquad \text{limit} \mapsto \lambda\kappa\delta.\mathcal{E}[\![\text{while value} < \text{bound do increment}]\!]([\text{bound} \mapsto \delta] \oplus \rho')\kappa]\\
\rho' &= [\text{this} \mapsto C_{this}] \oplus C_{attr}
\end{aligned}
$$

The inheritance of Counter to define StepCounter is demonstrated below. In these definition, the references to P in the denotation of a class must be bound to the components of Counter listed above, while C' is used to refer to the new class components. The local declarations of StepCounter consist of a variable step and a new implementation of increment, which must replace the old implementation because increment is virtual. These definitions appear in the C'_{def} component. In addition, the imperative is extended to initialize step.

$$
\begin{aligned}
C'_{imp} &= \lambda\vartheta.\lambda\kappa.\,update(update(\vartheta\kappa)(\iota_2, 1))(\iota_1, 0)\\
C'_{this} &= [\ \text{StepCounter} \mapsto C'_{attr}\\
&\qquad \text{Counter} \mapsto C_{attr}]\\
C'_{attr} &= S_{attr}|\{\text{increment}\} \oplus C'_{def}\\
\rho' &= [\text{this} \mapsto C'_{this}] \oplus C'_{attr}\\
C'_{def} &= [\ \text{step} \mapsto \iota_2,\\
&\qquad \text{increment} \mapsto \mathcal{E}[\![\text{value} := \text{value} + \text{step}]\!]\rho',\\
&\qquad \text{limit} \mapsto \lambda\kappa\delta.\mathcal{E}[\![\text{while value} < \text{bound do increment}]\!]([\text{bound} \mapsto \delta] \oplus \rho')\kappa]\\
C'_{virt} &= \{\text{increment}\}
\end{aligned}
$$

The fixpoint of the StepCounter class, which specifies the behavior of instances, is illustrated below:

$$
\begin{aligned}
C'_{imp} &= \lambda \vartheta . \lambda \kappa . \mathit{update}(\mathit{update}(\vartheta\kappa)(\iota_2, 1))(\iota_1, 0) \\
C'_{this} &= [\ \mathsf{StepCounter} \mapsto C'_{attr} \\
& \mathsf{Counter} \mapsto C_{attr}\,] \\
C'_{attr} &= [\ \mathsf{step} \mapsto \iota_2, \\
& \mathsf{increment} \mapsto \mathcal{E}[\![\mathsf{value} := \mathsf{value} + \mathsf{step}]\!]\rho', \\
& \mathsf{limit} \mapsto \lambda \kappa\delta . \mathcal{E}[\![\mathsf{while\ value} < \mathsf{bound\ do\ increment}]\!]([\mathsf{bound} \mapsto \delta] \oplus \rho')\kappa\,] \\
\rho' &= [\mathsf{this} \mapsto C'_{this}] \oplus C'_{attr}
\end{aligned}
$$

Chapter 8

Inheritance in Smalltalk

Smalltalk[11] may be understood as a simplification and reformulation of the basic concepts of Simula, adapted for use in an interpreted language. The fact that Smalltalkis a simpler language is reflected in its semantics, which is described naturally by generator combination.

Smalltalk introduced the concept of *metaclasses* to further the uniform application of the 'classes as objects' philosophy. The primary use of a metaclass is to define class variables, providing state that is shared by instances, and class methods, which are typically specialized constructors or instance initializers.

The semantics of Smalltalk is presented in two stages, which correspond roughly to the two stages in the development of Smalltalk, released in 1976 and 1980. The first phase introduces class inheritance, and the second phase demonstrates how essentially the same inheritance mechanism can be used to define the semantics of metaclasses. This analysis explains metaclasses in a uniform manner, exposing the symmetry between classes and metaclasses.

8.1 Smalltalk Without Metaclasses

8.1.1 Syntax

A simple syntax is introduced to represent Smalltalk programs without metaclasses. The informal meaning of the syntactic forms, and especially the scoping rules, are defined.

Programs: Prog ::= C P | E

Classes: Class ::= class I I′ V M

Expressions: Exp ::= new I | self | super

A program is a list of class definitions followed by an expression. The class definitions form a mutually recursive declaration, thus all class names are visible everywhere in the program.

A class declaration includes the name I of the class, the name of the parent class I′, the instance variables V, and the methods M. Instance variables are treated specially because they can be referenced in any descendant of the class in which they are defined. The class declaration may redefine any method defined in the parent, but instance variables cannot be redefined. Inheritance ensures that parent methods will invoke the redefined versions of methods.

The variables self and super are treated specially in the semantics. Self is used to access other methods declared in M, because the method names themselves are not directly included in the environment in which method expressions are evaluated. Super is used to refer to the methods declared by the parent class I′, in case a method is redefined.

The expression new I is used to create instances of the class I.

A coding of the counter example is given below, which closely resembles its encoding in Smalltalk. Comments have been inserted in italics to improve readability.

```
class Counter
inherit Base
variables
    value
methods
    proc increment()
        value := value + 1;
    proc limit(bound)
        while value < bound do
            self.increment();


class StepCounter
inherit Counter
```

Instance specifications	P, C	$\in$	**Inst**	=	**Env** $\times$ **Gen**
Instance continuations	ψ	$\in$	**InstCont**	=	**Cont(Inst)**
Classes	ζ	$\in$	**Cla**	=	**Cont(InstCont)**
Wrappers			**Wrap**	=	**Env**$\rightarrow$**Env**$\rightarrow$**Env**

Table 8.1: Semantic domains for Smalltalk.

```
variables
    step
methods
    proc increment;
        for i := 1 to step do
            super.increment;
```

8.1.2 Domains

The domains used for interpreting Smalltalk without metaclasses are defined in Table 8.1. Class denotations ζ are designed to provide for both creation of instances and inheritance. Their primary purpose, instance creation, is achieved by applying the class denotation to a continuation that expects a completed instance specification. An instance specification is a pair consisting of an environment that maps variables to their locations, and a generator of method behavior. The instance is derived by simply taking the fixpoint of the generator.

To achieve inheritance, the instance specifications must be extended. This is done by adding more instance variables and modifying the method behavior generator.

Since parent instance variables can be referenced in any descendant, an instance specification has an environment of parent variables in addition to the generator of parent functionality. If implicit access to parent variables were prohibited, as suggested by Snyder [25], then instance specifications could omit the parent variable environment, and the denotational semantics would be simplified to some degree, as in Chapter 6.

8.1.3 Semantics

A Smalltalk program is a mutually recursive binding of identifiers to class definitions. The program valuation $\mathcal{P}$ builds the generator of this recursive binding by combining the generators from each class definition. The expression which starts the computation is then evaluated in the environment built from this generator. The initial environment γ_i contains a definition for the class Base, which does not define any variables or methods.

$$\mathcal{P} : \mathbf{Prog} \rightarrow \mathbf{Gen} \rightarrow \mathbf{Cc}$$

$$\mathcal{P}[\![\mathsf{C\ P}]\!]\gamma = \mathcal{P}[\![\mathsf{P}]\!](\gamma \oplus_\perp \mathcal{C}[\![\mathsf{C}]\!])$$
$$\mathcal{P}[\![\mathsf{E}]\!]\gamma = \mathcal{E}[\![\mathsf{E}]\!](\mathrm{fix}(\gamma_i \boxed{\oplus_\perp} \gamma))(\mathit{print\text{-}answer})(\sigma_\emptyset)$$

$$\gamma_i : \mathbf{Gen_{Cls}}$$

$$\gamma_i = \lambda\rho\,.\,[\,\mathsf{Base} \mapsto \lambda\psi\,.\,\psi(\rho_\emptyset, \gamma_\emptyset)\,]$$

The valuation $\mathcal{C}$ specifies the meaning of a class definition. The environment in which a class is evaluated contains the denotation of all other classes in the program. The result of a class declaration is a little environment consisting of a binding for just the class being declared. The value of this binding is the composition of the allocator of the parent, taken from the class environment by $\rho(\mathsf{I}')$, with the local declarations of the class.

$$\mathcal{C} : \mathbf{Class} \rightarrow \mathbf{Gen}$$

$$\mathcal{C}[\![\mathsf{class\ I\ I'\ V\ M}]\!]\rho = [\,\mathsf{I} \mapsto \rho(\mathsf{I}') \circ \mathcal{A}[\![\mathsf{V\ M}]\!]\rho\,]$$

The valuation $\mathcal{A}$ defines the contribution of a class definition to the overall meaning of a class as a function on class specification continuations. It uses continuations because it must modify the store in allocating local variables.

The continuation argument ψ expects a specification of the new instance up to the level of this class definition; the continuation may extend the specification if it is part of a subclass, or it may simply build a complete instance from the specification. The process of extension is defined here, in the semantics of class definitions, while building an instance takes place in the semantics of the expression $[\![\mathsf{new\ I}]\!]$.

The result of the local declaration valuation $\mathcal{A}$ is a class specification continuation that takes the parent class variables ρ_p and method generator γ. These are extended with the locally defined variables and methods, by allocating the local variables with $\mathcal{V}$, and combining the local wrapper with the parent method generator. The complete set of local variables and the resulting generator are passed to ψ.

$$\boxed{\begin{array}{l} \mathcal{A} : \textbf{Var} \times \textbf{Proc} \to \textbf{Env} \to \textbf{InstCont} \to \textbf{InstCont} \\ \\ \mathcal{A}[\![\mathsf{V}\ \mathsf{M}]\!]\rho\psi = \lambda(\rho_p, \gamma_p) . \textbf{clet } \rho_v = \mathcal{V}[\![\mathsf{V}]\!] \textbf{ in} \\ \qquad\qquad\qquad\qquad \textbf{let } \rho' = \rho_v \oplus_\perp \rho_p \textbf{ in} \\ \qquad\qquad\qquad\qquad\quad \psi(\rho', \mathcal{W}[\![\mathsf{M}]\!](\rho' \oplus \rho) \boxed{\triangleright} \gamma_p) \end{array}}$$

The methods of the class are converted into a wrapper by the valuation $\mathcal{W}$ as defined in Section 2.4.2. The environment ρ_s represents self-reference within the methods, and is bound to **self**, while ρ_p represents parent methods, and is bound to **super**.

Note that the methods are evaluated in an environment in which only **self**, **super**, the instance variables, and the class identifiers are bound, *but not the names of the other methods*. This is because the normal evaluation $\text{fix}(\mathcal{M}[\![\mathsf{M}]\!])$ of a mutually recursive function binding is not used; instead the generator converted so that **self** is used to access the recursive fields, using the construct $\text{fix}(\lambda\rho . \mathcal{M}[\![\mathsf{M}]\!]([\mathsf{self} \mapsto \rho]))$

$$\boxed{\begin{array}{l} \mathcal{W} : \textbf{Proc} \to \textbf{Env} \to \textbf{Wrap} \\ \\ \mathcal{W}[\![\mathsf{M}]\!]\rho = \lambda\rho_s . \lambda\rho_p . \mathcal{M}[\![\mathsf{M}]\!]([\mathsf{self} \mapsto \rho_s, \mathsf{super} \mapsto \rho_p] \oplus \rho) \end{array}}$$

The **new** expression constructs an instance from the specification produced by invoking a class denotation. The instance variables are discarded, and the fixpoint of the method generator is used to denote the new instance.

$$\boxed{\begin{array}{l} \mathcal{E}[\![\mathsf{new\ I}]\!]\rho\kappa = \textbf{clet } \rho', \gamma = \rho(\mathsf{I}) \textbf{ in} \\ \qquad\qquad\qquad \kappa(\text{fix}(\gamma)) \end{array}}$$

8.1.4 Example

The semantics is illustrated by sketching the interpretation of the **Counter** example given in Section 8.1.1. The valuation of this program example results in an environment ρ_c in which the symbols **Counter** and **StepCounter** are bound. The structure of this environment is illustrated below. Letting M_1 represent that declaration of methods in the class, the expression bound to Counter would be:

$$\begin{array}{l}\rho_c(\mathsf{Counter})\\ \quad = \lambda\psi\,.\,\mathbf{clet}\ \rho_v = \mathcal{V}[\![\mathsf{value}]\!]\ \mathbf{in}\\ \qquad\qquad \psi(\rho_v, \mathit{bind}_{\mathsf{self}}\mathcal{M}[\![\mathsf{M}_1]\!]\rho_v)\end{array}$$

This simply means that Counter is bound to a function that takes a continuation, allocates a variable value and then passes this variable and a generator of the methods in M_1 to the continuation.

The function $\mathit{bind}_{\mathsf{super}}$ is used to convert a generator which expects the parent record to be bound to the symbol super, as described in the previous section, into a wrapper:

$$\mathit{bind}_{\mathsf{sym}}G = \lambda s\,.\,\lambda p\,.\,G([\mathsf{sym} \mapsto p] \oplus s)$$

The second class declaration inherits from the first: it defines a continuation that to which the Counter denotation is applied. This continuation allocates another variable, step, and then combines the generator from Counter with the wrapper specified by the local methods. The binding of StepCounter is yet another continuation function, similar to the one given above, to which the resulting variable environment and wrapped generator are passed. M_2 represents the declaration of methods in StepCounter.

$$\begin{array}{l}\rho_c(\mathsf{StepCounter}) =\\ \quad \lambda\psi\,.\ \rho_c(\mathsf{Counter})\\ \qquad\quad (\lambda(\rho_p,\gamma_p)\,.\,\mathbf{clet}\ \rho_v = \mathcal{V}[\![\mathsf{step}]\!]\ \mathbf{in}\\ \qquad\qquad\qquad \mathbf{let}\ \rho' = \rho_v \oplus_\perp \rho_p\ \mathbf{in}\\ \qquad\qquad\qquad\quad \psi(\rho', \mathit{bind}_{\mathsf{self}}(\mathit{bind}_{\mathsf{super}}\mathcal{M}[\![\mathsf{M}_2]\!])\rho_v\ \boxed{\rhd}\ \gamma_p))\end{array}$$

By expanding the value of ρ_c(Counter), and essentially equivalent class denotation can be given that does not involve inheritance:

$$\begin{array}{l}\rho_c(\mathsf{StepCounter}) =\\ \quad \lambda\psi\,.\,\mathbf{clet}\ \rho_v = \mathcal{V}[\![\mathsf{value{,}step}]\!]\ \mathbf{in}\\ \qquad\quad \psi(\rho_v, \mathit{bind}_{\mathsf{self}}(\mathit{bind}_{\mathsf{super}}\mathcal{M}[\![\mathsf{M}_2]\!])\rho_v\ \boxed{\rhd}\ \mathit{bind}_{\mathsf{self}}\mathcal{M}[\![\mathsf{M}_1]\!]\rho_v)\end{array}$$

8.2 Smalltalk with Metaclasses

8.2.1 Metaclasses

In Smalltalk-80, *metaclasses* were introduced to enable the uniform interpretation of classes as objects: metaclasses represented the class of a class. The result was a useful (though sometimes confusing) symmetry. They also solved the practical problem of object initialization that was caused by dropping the imperative of Simula. The **new** statement of Simula became a message sent to a class object requesting the creation of a new instance. Since all messages are handled by the class of the receiver, the metaclass would handle the **new** message, and could perform initialization after creating a basic instance. Metaclasses also specify the local 'class' variables of their instance (which is a class).

Metaclasses have exactly one instance, so there is a one-to-one correspondence between classes and metaclasses. In fact, a class and its associated metaclass are defined together in a single construct. Though carefully fitted into a symmetric framework, metaclasses and classes do not actually have much in common as far as intent and purpose.

8.2.2 Metaclass Syntax

The syntax for a smalltalk class and its associated metaclass is a simple extension of the syntax of a simple class:

Classes: Class ::= class I I' V_m M_m V_i M_i

The additional variables V_m and methods M_m represent class variables and class methods which define metaclass behavior.

8.2.3 Metaclass Semantics

A metaclass is just a *class constructor*. The metaclass contains all structures necessary to create instances of the class, hence the **new** expression and the class generator become components of the metaclass. The generator is stored in the metaclass as a special attribute named **inst**. The **new** method has the effect of creating an instance by instantiating **self.inst**.

Unlike normal constructors, a metaclass is used only once to construct a single class. Thus each class variable specified in M_1' is allocated once and shared by all subclasses of the class. A metaclass is like a class that is immediately instantiated and in addition has a inst field that is handled specially. The template must be expanded in an environment in which all class variables are defined, and then be combined with inheritance to the template of the super-metaclass. Thus two levels of inheritance must occur: inheritance of metaclass methods and variables, and then inheritance of class variables and methods.

A class/metaclass declaration is elaborated by first accessing the parent class specification from the class environment. The class definition is then examined to produce a new class variable environment and class wrapper. The wrapper is combined with the parent generator and passed to the declaration continuation.

$$\mathcal{C} : \textbf{Class}\to\textbf{Env}\to\textbf{Cont(Env)}\to\textbf{Cc}$$

$$\begin{aligned}\mathcal{C}'[\![\mathsf{class\ I\ I'\ V}_m\ \mathsf{M}_m\ \mathsf{V}_i\ \mathsf{M}_i]\!]\rho\kappa = {} & \textbf{let } (\rho',\gamma) = \rho(\mathsf{I}') \textbf{ in} \\ & \quad \textbf{clet } (\rho'',\omega) = \mathcal{A}'[\![\mathsf{V}_m\ \mathsf{M}_m\ \mathsf{V}_i\ \mathsf{M}_i]\!]\rho' \textbf{ in} \\ & \quad\ \kappa[\mathsf{I} \mapsto (\rho'',\omega \boxed{\triangleright} \gamma)]\end{aligned}$$

The metaclass aspects of the declaration are handled by allocating the locally defined variables and then producing a wrapper for the methods. The class specific declarations are simply handled within the wrapper.

$$\mathcal{A}' : \textbf{Var}\times\textbf{Proc}\times\textbf{Var}\times\textbf{Proc}\to\textbf{Env}\to\textbf{InstCont}\to\textbf{InstCont}$$

$$\begin{aligned}\mathcal{A}'[\![\mathsf{V}_m\ \mathsf{M}_m\ \mathsf{V}_i\ \mathsf{M}_i]\!]\rho\chi = {} & \textbf{clet } \rho' = \mathcal{V}[\![\mathsf{V}_m]\!] \textbf{ in} \\ & \textbf{let } \rho'' = \rho' \oplus \rho \textbf{ in} \\ & \quad \chi(\rho'', \mathcal{W}'[\![\mathsf{M}_m\ \mathsf{V}_i\ \mathsf{M}_i]\!]\rho'')\end{aligned}$$

The metaclass wrapper is the same as a class wrapper except for the addition of an inst template, whose semantics is directly analogous to that of classes defined above. The double-boxed combination function $\boxed{\boxed{\oplus}}$ is used to combine wrappers in this case, by distributing self and super to both of them.

$$\mathcal{W}' : \textbf{Proc}\times\textbf{Var}\times\textbf{Proc}\to\textbf{Env}\to\textbf{Wrap}$$

$$\mathcal{W}'[\![\mathsf{M}_m\ \mathsf{V}_i\ \mathsf{M}_i]\!]\rho = \mathcal{W}[\![\mathsf{M}_m]\!]\rho \boxed{\boxed{\oplus}} \lambda\rho_s\rho_p.\,[\mathsf{inst} \mapsto \rho_p(\mathsf{inst}) \circ \mathcal{A}[\![\mathsf{V}_i\ \mathsf{M}_i]\!]\rho])$$

The basicNew method is defined in class Base, which serves as the starting-point for inheritance:

$$\mathsf{Base} \mapsto [\mathsf{inst} \mapsto \lambda\psi.\psi(\lambda\rho.[\mathsf{basicNew} \mapsto \lambda\kappa.\rho(\mathsf{inst})\{\lambda(\rho',\gamma).\kappa(\mathrm{fix}(\gamma))\}])]$$

The regularity of the treatment of metaclass and class inheritance shows that the two concepts are very similar. The metaclass mechanism generalizes to higher levels of metaclassing. A meta-metaclass would be one that created metaclasses, and would handle the messages sent to meta-classes. All that is needed is for yet another layer template within what is now considered the instance level. This template would be combined with the corresponding template from a super-object, and a new method provided to instantiate the template.

The uniformity of Smalltalk-80, though in many ways a simplification of Simula, has a tendency to confuse novice programmers. A review [5] of Smalltalk-80, based on intuitive and empirical considerations, produced the recommendation that metaclasses be eliminated, reverting to the approach in Smalltalk-76. However, other 'confusing' aspects of Smalltalk-80, self and super in particular, were retained on the grounds of the evident utility, though no logical justification was offered. By examining the semantics of Smalltalk-80 class declarations, it is clear that metaclasses are a complicated construct, for which alternative solutions exist.

Chapter 9

Inheritance in Beta

In Beta[15, 16], the direct successor of Simula, the notion of class prefixing is generalized to allow attributes to be prefixed. In Beta a class is called a *pattern*, which has attributes and an imperative. The attributes of a pattern may be locations which refer to pattern instances or local pattern definitions. Inheritance of patterns is achieved by *prefixing* as in Simula. But inheritance is extended to pattern attributes, allowing the parts of a pattern to be extended as the complete pattern is inherited. Attribute prefixing resembles method replacement in other object-oriented languages but it is more controlled because it uses the prefixing discipline instead of complete replacement. The **inner** construct is uniformly extended for use within the imperative of a pattern attribute to invoke the corresponding sub-pattern attribute imperative. Qualification and **this** are not used in Beta.

These changes give Beta a stronger form of definitional consistency, because the behavior of a pattern cannot be changed radically during inheritance, but only extended in a controlled fashion, mediated by **inner**.

9.1 Syntax

The language used to characterize Beta inheritance is defined in Table 9.1. It omits static references, concurrency, arrays, strong typing declarations, selection of pattern attributes from uninstantiated patterns, etc. These changes do not have much real impact on the fundamental semantic nature of inheritance in the language. The syntax is explained below.

Program			**Prog**	::=	P
Patterns	P	∈	**Pat**	::=	I (# V D #) |
Declarations	D	∈	**Dcl**	::=	I : P | enter $\vec{\mathsf{I}}$ do E
Expressions	E	∈	**Exp**	::=	inner |

Table 9.1: Abstract Beta syntax.

A program is simply a pattern definition, whose imperative is executed to perform the top-level computation.

A pattern declaration consists of a prefix I and an *object description* (# V D#), which is also called the *suffix*. The prefix names the pattern that is to be extended with the additional features in the object description. As usual, the predefined pattern Base with no attributes serves as the root of the inheritance hierarchy. The object description defines state variables V and a collection of *pattern attributes* D. The pattern attributes are analogous to methods in other languages.

The pattern attributes and variables defined by the prefix are inherited, and combined with the local declarations. First, any use of inner within the prefix attributes or imperative is made to refer to the corresponding component of the object description. Then, the attributes of the new pattern are determined by taking first from the pattern attributes of the prefix, and then including any additional attributes in the object description not defined in the prefix. The result takes its imperative from the prefix. Finally, all references to components from within the prefix or the object description are made to refer to the combined pattern component.

This allows a form of attribute prefixing in which the attributes in the prefix are automatically combined with those in the suffix. This form of attribute prefixing is slightly different from the actual Beta syntax, in which the complete definition of the pattern must be given in the suffix. This notation is avoided here because it has the connotation that the entire attribute of the prefix is being replaced by the complete new definition provided by the suffix, even though the new definition must be an extension of the original.

Real Beta also has additional syntax for determining when attributes are virtual and can be redefined, signaling that a redefinition is being made, and preventing any further redefinition. In the subset of Beta examined here, all attributes are assumed to be virtual, so none of these indications are necessary. Allowing non-virtual attributes would require some change in this presentation, adopting some features from the Simula semantics in Section 7, but would add little insight.

Declarations are used to specify pattern attributes and the imperative. The treatment of the imperative deserves some discussion. The imperative is included as a declaration among the pattern attributes because its behavior with respect to inner is similar. The imperative will be converted into a functional component with the special name imp, which is executed when the pattern is instantiated. During prefixing the imperative is treated exactly like the pattern attributes. The novel notation of 'flow expressions' used for expressions in Beta is not described here, because the difference is primarily cosmetic.

The special expression inner may appear in the imperative of a pattern, and indicates that the corresponding sub-pattern imperative, if defined, should be executed at that point in the prefix imperative.

A pattern is instantiated by naming it, causing the execution of its imperative. This expression returns the value of the imperative execution. References are not currently supported in this analysis.

Interestingly, the Counter example cannot be expressed in Beta in the same way as in the other object-oriented languages. This is because Beta has a strong notion of definitional consistency, so that the increment attribute of the parent class cannot be simply replaced by an iterating version. Inheritance in Beta is intended to be incremental, so modifications can only be made by adding to the behavior of inner.

```
Counter : Base (#
    var value;

    increment : (#
        enter do
            value := value + 1;
            inner;
        #);

    limit : (#
        enter bound do
            while value < bound do
                increment();
                inner;
        #);

    enter do
        value := 0
        inner
    #)

ModCounter : Counter (#
    var step;

    increment : (#
        enter do
            inner;
            if value mod step ≠ 0 then
        increment;
    #);

enter do
    step := 1;
    inner
#)
```

This example demonstrates how behavior can only be added to an imperative during inheritance. Since the original **increment** attribute cannot be replaced by **ModCounter**, it is essential that it be written to invoke the inner method at an appropriate place.

The extension of **increment** in **ModCounter** takes advantage of the fact that it is called after the value is incremented to check if it is an even multiple of **step**. If not, it calls **increment** to test again; thus it is tail-recursive.

Note that **inner** was also inserted in a likely place in the definition of **limit**. The location of **inner** must be guided by a prediction of what kind of extensions might be useful, and what kind should be allowed.

9.2 Domains

The domains used for the semantics of Beta are structurally the same as those used in Smalltalk, as defined in Section 8.1.2. However, there is in a difference in the kinds of values in the environment of the generator. In Smalltalk the generator component of an instance specification is a simple generator of functional environments of the form $\mathbf{Env}_{\mathbf{Fun}} \rightarrow \mathbf{Env}_{\mathbf{Fun}}$. In Beta each attribute in the result of the generator is a function of the corresponding **inner** attribute, and selective combination is used to match these components. Thus a Beta instance specification has the form $\mathbf{Env}_{\mathbf{Fun}} \rightarrow \mathbf{Env}_{\mathbf{Fun} \rightarrow \mathbf{Fun}}$.

In addition, the special symbol **imp** is used to indicate the imperative of the pattern. Its type in the generator is different, being based upon **Cmd** instead of **Fun**.

9.3 Semantics

In Beta, the pattern is essentially a *functional abstraction* or generic definition, the actual parameter being supplied by a sub-pattern. The keyword **inner** represents this implicit formal parameter in the prefix definition. When the pattern is instantiated by itself, the parameter is filled by an empty sub-pattern. Prefixing is essentially composition of these generics.

The semantics of **inner** is given by *selective inheritance* (see Section 2.4.3), because **inner** in an attributes is automatically associated with the corresponding attributes of the sub-pattern. Since prefixing is uniform with respect to pattern attributes and the imperative, simple composition serves to form prefixed sub-patterns and to resolve inner imperatives. The semantics of patterns is given below:

$$\mathcal{C} : \mathbf{Pat} \rightarrow \mathbf{Env} \rightarrow \mathbf{Dc} \rightarrow \mathbf{Dc}$$

$$\mathcal{C}[\![\mathsf{I}\ (\#\ \mathsf{V}\ \mathsf{D}\ \#)]\!]\rho = \rho(\mathsf{I}) \circ \mathcal{A}[\![\mathsf{V}\ \mathsf{D}]\!]\rho$$

$$\mathcal{A} : \mathbf{Dcl} \times \mathbf{Dcl} \rightarrow \mathbf{Env} \rightarrow \mathbf{InstCont} \rightarrow \mathbf{InstCont}$$

$$\begin{aligned} \mathcal{A}[\![\mathsf{V}\ \mathsf{D}]\!]\rho\psi = \lambda\, \rho_p, \gamma_p \,.\, \mathbf{clet}\ \rho_v &= \mathcal{V}[\![\mathsf{V}]\!]\ \mathbf{in} \\ \mathbf{let}\ \rho' &= \rho_v \oplus_\bot \rho_p\ \mathbf{in}\ \psi(\rho', \gamma_p \boxed{\oplus_\circ} \mathcal{W}[\![\mathsf{D}]\!](\rho' \oplus \rho)) \end{aligned}$$

$$\mathcal{W} : \mathbf{Dcl} \rightarrow \mathbf{Wrap}$$

$$\mathcal{W}[\![\mathsf{D}]\!]\rho = \lambda\rho'.\mathcal{M}[\![\mathsf{D}]\!]((\rho'|\mathcal{B}[\![\mathsf{D}]\!]) \oplus \rho)$$

Note that if a variable is in a position to be redefined, an error will be generated during composition, since variables are bound to locations, not composable functions.

Declarations are changed to allow definition of constant patterns, and to convert the imperative into a procedure.

$$\mathcal{D} : \mathbf{Dcl} \rightarrow \mathbf{Env} \rightarrow \mathbf{Dc} \rightarrow \mathbf{Cc}$$

$$\begin{aligned}
&\mathcal{D}[\![\mathsf{I} : \mathsf{P}]\!]\rho = [\mathsf{I} \mapsto \mathcal{P}[\![\mathsf{P}]\!]\rho] \\
&\mathcal{D}[\![\mathsf{enter}\ (\vec{\mathsf{I}})\ \mathsf{do}\ \mathsf{E}]\!]\rho = [\mathsf{imp} \mapsto \lambda\vartheta.\lambda\kappa.\lambda\vec{\delta}.\mathcal{E}[\![\mathsf{E}]\!]([\vec{\mathsf{I}} \mapsto \vec{\delta}, \mathsf{inner} \mapsto \vartheta] \oplus \rho)\kappa]
\end{aligned}$$

To execute a pattern, it is instantiated to produce an environment generator, passing in an empty environment $\langle\rangle$ to represent the fact that no prefixing is to occur. The imperative is selected from the environment, and passed a no-op (represented by $\vartheta_\emptyset$) to clear the effect of inner.

```
$\mathcal{E}'[\![\mathsf{I}]\!]\rho\kappa$ = clet $\delta = \mathcal{E}[\![\mathsf{I}]\!]\rho$ in
                 if $\delta \in \mathbf{Inst}$
                 then clet $\rho, \gamma = \rho(\mathsf{I})$ in
                          let $\delta = \mathrm{fix}(\gamma \boxed{\oplus.} \gamma_{\mathrm{null}})$ in
                             $\delta(\mathsf{imp})\kappa\langle\rangle$
                 else $\kappa(\delta)$
$\gamma_{\mathrm{null}} = \lambda s.\,[\mathsf{imp} \mapsto \vartheta_\emptyset,\ \mathit{others} \mapsto \gamma_\emptyset]$
```

9.4 Beta Compared with Smalltalk

The semantics of Beta and Smalltalk are very similar; however, they differ in the central mechanism used to combine the inherited structure with the local definitions. The following table brings this difference into focus. γ_p is the parent generator, ρ' is an environment of variables, ρ is the global environment.

Smalltalk :	$\mathcal{W}_{Smalltalk}[\![\mathsf{D}]\!](\rho' \oplus \rho) \boxed{\triangleright} \gamma_p$
Beta :	$\gamma_p \boxed{\oplus_\circ} \mathcal{W}_{Beta}[\![\mathsf{D}]\!](\rho' \oplus \rho)$

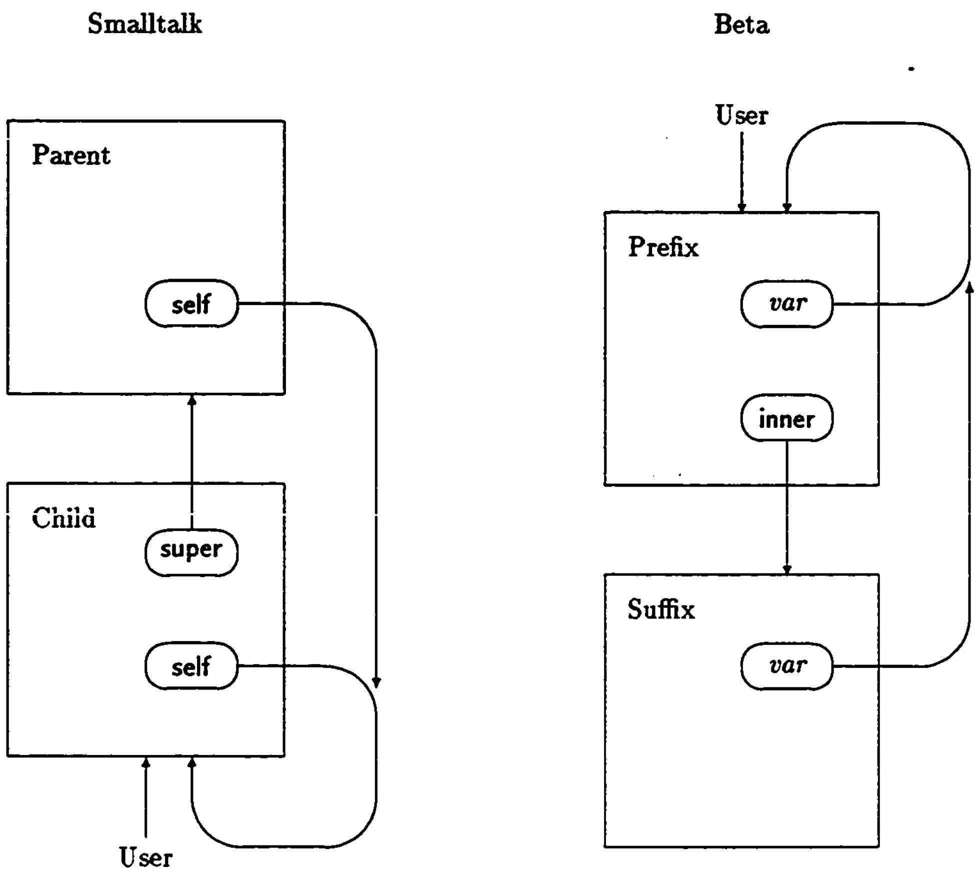


Figure 9.1: Inverse hierarchies in Smalltalk and Beta.

The primary difference is that the parent generator is on the right in Smalltalk but on the left in Beta. This means that Beta attributes cannot be replaced. The other difference is that Smalltalk uses $\boxed{\triangleright}$ to apply the changes, while Beta uses $\boxed{\oplus_o}$ giving selective composition.

The surprising conclusion to be drawn from this picture is that Beta has an inverted notion inheritance from other object-oriented languages. In Beta, a *super*-pattern acts like a *sub*-class, while a *sub*-pattern acts as a *super*-class. This is because the super-pattern generator is composed to the *left* of the sub-pattern, thus taking precedence over it. The **inner** statement corresponds more closely to **super** than **self**. The notion of self-reference is implicit, being an artifact of mutually recursive definitions. This situation is illustrated in Figure 9.1. Note that self has been replaced by an explicit variable reference *var* in Beta.

Chapter 10

Inheritance in Flavors

Flavors is an object-oriented extension of Lisp. The most significant advance from Flavors was the introduction of *multiple inheritance.* However, multiple inheritance in Flavors is fundamentally different from multiple inheritance as presented in Section 2.5.1, because it is based upon *linearization.* Though the process of linearization is potentially detrimental to the encapsulation of programs [25], the conventional technique of *mixins* depends directly on the violation.

This chapter focuses on the semantics of linearization, in such a way that problems inherent in linearization are made clear. In addition the use of linearization for mixing is investigated, and a more modest approach which does not have the potential to violate encapsulation is suggested. The mechanism of *method combination* is also examined.

10.1 Syntax

In choosing a simple language to represent Flavors inheritance, the intent is to include the most regular and powerful notations, and omit notations that are covered by more general forms. This abbreviated syntax is given in Table 10.1.

Flavors			**Dcl**	::=	(defflavor I($\vec{\mathsf{I}_v}$) ($\vec{\mathsf{I}_p}$)
					(defmethod M_m)
					(defwrapper M_w)
Expressions	E	$\in$	**Exp**	::=	new I

Table 10.1: Flavors syntax.

The declaration of a flavor specifies the flavor name I_N, a list of instance variables $\vec{I}_v$, and a list of inherited flavors $\vec{I}_p$. The ancestors of a flavor consist of all the inherited flavors and their ancestors. The associated **defmethod** form specifies the main methods of the flavor, and the **defwrapper** form defines any wrappers.

The instances of a flavor have local state represented by the union of all the variables declared in the flavor and all its ancestors. When more than one ancestor uses the same variable name, only a single variable is present in the instance, and both ancestors will refer to the same storage location. Thus instance variables are shared among ancestors.

The behavior of a flavor is defined by a complex combination of the main methods and wrappers that appear in the flavor and its ancestors.

The variable **self** is given a special binding just as in Smalltalk. The variable **continue** is given a special binding in the evaluation of the wrappers. This symbol is bound to a function that represents the 'main-method' that is being wrapped. Thus, a typical wrapper will perform some initial computations, then invoke **continue** to perform the main method, and then finish the computation. **Continue** can only be used to invoke the same method of the parent, and Flavors does have the more general **super** construct of Smalltalk, which can access arbitrary parent methods.

To determine the behavior of an instance, the ancestor graph is first linearized. There are several schemes for linearization, but the general idea is to perform a left-to-right depth-first search up to duplicates of the ancestor graph of the flavor. Thus the order in which the inherited flavors are listed in the flavor definition may an impact on the behavior of instances. The effect of linearization is to form a linear list of flavor definitions in which duplicates have been removed, from which the behavior of the class is easily constructed.

Given the linearized list of flavors, the main methods are found by searching the list in order to find the first definition of each method. In addition, the wrappers are composed in order, so that **continue** in an earlier one will invoke the next wrapper in sequence, and **continue** within that one will invoke the next, etc. Finally, the wrappers are composed with the main methods, so that **continue** in the last wrapper will invoke the corresponding main method.

The *before* and *after* methods of Flavors have been omitted because their functionality can be achieved using wrappers.

```
(defflavor Counter
        (value) ; variables
        (Base) ; parent

    (defmethod (increment Counter) ()
        value := value + 1
        )

    (defmethod (limit Counter) (bound)
        while value < bound do
            self.increment()
        ))

(defflavor StepMixin
        (step) ; variables
        (Base) ; parent

    (defwrapper (increment StepMixin) ()
        for i := 1 to step do
            super.increment
        ))

(defflavor StepCounter
        () ; variables
        (StepMixin Counter)) ; parents
```

10.2 Domains

In the semantics of Flavors, there is a significant difference between the behavior of flavor instances and the behavior inherited by subflavors. Inheritance depends upon the complete *structure* of ancestor hierarchy. It is not enough to simply use the behavior (or generator of behavior) of its parents. Thus the 'denotation' of a flavor is merely a a tree containing the behavioral contributions of its parents.

The specification of a flavor is given by an environment of wrappers, an environment of methods, and a set of variables. These domains are summarized in Table 10.2.

Flavors	$\mathbf{Flav}$ =	[
Variables		$var: \mathbf{Ide}^*$
Main methods		$meth: \mathbf{Gen}_{\mathbf{Fun}}$
wrappers		$wrap: \mathbf{Gen}_{\mathbf{Fun}\rightarrow\mathbf{Fun}}$
Parents		$parent: \mathbf{I}\rightarrow\mathbf{Flav}$
		]

Table 10.2: Semantic domain of Flavor definition.

The denotation of a flavor is a tree structure which represents the behavioral contribution of this flavor together with all the behavioral contributions of its ancestors. The domain of Flavor denotations is thus

The wrapper environment contains *wrapper methods*, which are functions on denotable values. The argument to a wrapper method is the corresponding value in the parent, i.e. the value with the same name in the record of parent methods. The wrapper environment is assumed to be defined for all identifiers, with the identity function standing in for any unspecified wrappers. This convention allows wrapper environments to be composed without lose of information, but does not allow a wrapper to actually *define* an identifier if the parent base method is undefined.

Because the list of flavor components contained in a flavor may be arbitrarily rearranged when the flavor is inherited, it is not meaningful to say that the flavor behavior is inherited. Rather, the tree of ancestor components that is inherited. Also, since linearization may insert the components of a flavor directly into the middle of a linear order, a flavor that does not inherit from anything typically ends up having 'parents.' In this position it may act as a wrapper. This effect is relied upon heavily in flavors programming, through this use of *mixin*, which are by convention placed in front of the flavor they modify or wrap.

10.3 Semantics

Since the semantics of a flavor is an ancestor tree, the interpretation of a flavor definition simply builds a new ancestor tree by joining the parents ancestors together into a new tree.

The semantics for the flavor declaration given above is:

$\mathcal{F}$: **Dcl→Env→Flav**

$$\begin{aligned}
&\mathcal{F}[\![\text{(defflavor I}(\vec{\mathsf{I}_v})\ (\vec{\mathsf{I}_p}))\ \text{(defmethods M}_m)\ \text{(defwrappers M}_w)]\!]\rho \\
&\quad = [\mathsf{I} \mapsto [\ \ \mathit{var} \mapsto \vec{\mathsf{I}_v}, \\
&\qquad\qquad \mathit{meth} \mapsto \mathcal{M}[\![\mathsf{M}_m]\!], \\
&\qquad\qquad \mathit{wrap} \mapsto \mathcal{W}[\![\mathsf{M}_w]\!], \\
&\qquad\qquad \mathit{parent} \mapsto [\vec{\mathsf{I}_p} \mapsto \rho(\vec{\mathsf{I}_p})]\,] \\
&\qquad\quad] \\
&\mathcal{W}[\![\mathsf{M}]\!] = \lambda\rho.\lambda x.(\lambda a.\mathcal{M}[\![\mathsf{M}]\!]([\text{continue} \mapsto a] \oplus \rho))
\end{aligned}$$

It is impossible to combine these behaviors into a less syntactic "denotation" because the internal structure of the tree is important when inheritance behavior is derived.

The instance behavior of a flavor, on the other hand, is derived by combining the ancestor flavors. The instance behavior of a flavor definition is specified by a series of transformations. First, the tree of parent flavors is converted into a list by linearization. This linear list is then reduced to a single specification by composition and method combination. This final specification is used to instantiate the flavor instances.

Linearization serves to convert the tree structure of ancestors into a linear list in which duplicates are removed. The removal of duplicates ensures that each behavior defined by an ancestral flavor is performed only once, no matter how many times it appears in the ancestor tree. In Flavors, the rules governing linearization are

1. A flavor always precedes its components.
2. The local ordering of components is preserved.
3. Duplicate flavors are eliminated from the ordering.

A function *linearize* : **Flav→Flav*** is assumed to convert of a flavor into a list of flavors according to these rules, based on the flavor denotations in an environment. Though a flavor definition may not have a linearization that satisfies the rules, this possibility is ignore here as a minor complication.

The second stage is achieved by *iterated single inheritance.* Iterated single inheritance works by considering the linear list as a sequence of wrappers, each of which modifies and augments the behavior specified by the rest of the list. Iterated single inheritance is simply a composition function for flavors:

$$
\begin{array}{l}
\textit{compose} : \mathbf{Flav}^{*} \rightarrow \mathbf{Flav} \\
\\
\textit{compose}(F_1, \ldots, F_n) = [\\
\quad \mathit{var} = F_{1,\mathrm{var}} \cup \cdots \cup F_{n,\mathrm{var}} \\
\quad \mathit{wrap} = F_{1,\mathit{wrap}} \boxed{\oplus_\circ} \cdots \boxed{\oplus_\circ} F_{n,\mathit{wrap}} \\
\quad \mathit{meth} = F_{1,\mathit{meth}} \boxed{\oplus} \cdots \boxed{\oplus} F_{n,\mathit{meth}} \\
\quad]
\end{array}
$$

It is this behavior that is instantiated to produce a flavor instance. A behavior is instantiated allocating storage for the variables, then wrapping the wrapper methods around the main methods:

$$
\begin{array}{l}
\mathcal{E}[\![\mathsf{new}\ \mathsf{E}]\!]\rho\kappa = \mathbf{clet}\ F = \mathcal{E}[\![\mathsf{E}]\!]\rho\ \mathbf{in} \\
\qquad\qquad\qquad \mathbf{clet}\ \rho_v = \mathcal{V}[\![F_{\mathrm{var}}]\!]\ \mathbf{in} \\
\qquad\qquad\qquad\quad \kappa(\mathrm{fix}(\lambda\rho_s \,.\, (F_{\mathrm{wrap}} \boxed{\oplus_\cdot} F_{\mathit{meth}})([\mathsf{self} \mapsto \rho_s] \oplus \rho_v)))
\end{array}
$$

To create an instance, the variables of the flavor are instantiated, the wrappers are applied to the main methods, and the fixpoint of the resulting method is taken.

The base flavor Base, which serves as the parent of all Flavors, defines no variables, wrappers, or methods.

10.4 Method Combination

Multiple inheritance is used to form a new class by inheriting attributes from several parent classes at once. In order to handle the inevitable problem of conflicting attributes among several parents, Flavors introduced *method combination* to automatically construct derived methods. Method combination allows a method to be constructed from the pieces of functionality contributed by each parent. Method combination requires some uniformity in the way these method pieces are defined, especially in the way they interact parent methods.

Method combination gives additional flexibility and power to the Flavors language. With method combination, the main method of a subflavor need not simply replace the corresponding method in the parent. The subflavor method may be *combined* with parent methods. The form of combination is specified statically, and may involve logical combination, list concatenation, summation, etc.

Method combination is represented by a record of combination functions, which is part of the behavior of every flavor. Appropriate syntax should be added to the definition of Flavors. The method combination specification, *comb* : **Env**, is added as a component of the domain **Beh**. Method combination functions are unioned very much like instance variable names. The only effect of method combination is to change the way in which main method generators are combined. Instead of simply replacing all redefined methods with $\oplus$, each method may be individually combined with its parents, as specified by the combination function environment.

In this way each all methods under a single label are combined according to the corresponding function from the combination environment. combination function. Of course, the combination functions should not be require both arguments to be defined.

10.5 Linearization and Mixins

Since linearization reduces multiple inheritance to iterated single inheritance, the parents that are placed first in the list end up "wrapping" those placed later in the list, even though they may not have originally specified this relationship. This effect is controversial because it provides added functionality that is desired, but at the cost of making complex inheritance hierarchies very difficult to understand.

Though languages with linearization do not support an explicit wrapper construct[1], they achieve the effect of wrappers by depending upon linearization to wrap the first parents around later ones [26]. Adding an explicit wrapping construct to these languages would obviate the need for linearization, and at the same time greatly simplify the behavior of multiple inheritance.

[1]Wrappers in Flavors [20] are used to wrap a particular method. They are not a record wrapping construct.

Chapter 11

Related Work

11.1 Cardelli

Cardelli [6] proposed a semantics of inheritance in which he identified inheritance with subtyping. His notion of record subtyping was presented in Chapter 4. Cardelli claimed that the subtype relation on record types "corresponds to the *subclass* relation of Simula and Smalltalk." He illustrated this correspondence by the following examples:

```
type any      =  ()
type object   =  (age: int)
type vehicle  =  (age: int, speed: int)
type machine  =  (age: int, fuel: string)
type car      =  (age: int, speed: int, fuel: string)
```

Since car has all the properties of vehicle, car is a subtype of vehicle. Cardelli's interpretation was that "a car has (inherits) all the attributes of vehicle and of machine." This is also an example of *multiple inheritance* because the car type is a subtype of (inherits from) two types: vehicle and machine.

Cardelli points out that the attributes in the records types listed above resemble the instance variables of an object in object-oriented programming, while these objects typically consist of methods. Cardelli addresses this point by using functions as the values in a record.

In a later paper [7], Cardelli and Wegner point out that since the functions in a record are not specified in a record type (only their type is given), objects must be created explicitly, independently of their notion of inheritance. One effect of this is that "at record creation time one must choose explicitly which field values a particular record should have: whether it should *inherit* them by using some predefined function (or value) used in the allocation of other records, or *redefine* them by using a new function (or value). Everything is allowed as long as the type constraints are respected." They also note that method refinements using "the concept of super ... cannot be simulated because they imply an explicit class hierarchy."

Cardelli provides two examples of explicit object creation. The examples are expressed in a language similar to ML with updatable reference variables. The first example illustrates the use of self-reference:

```
type point = (x:real, y:real)
type active_point = point and (d: point → real)
val make_active_point(px:real, py:real) : active_point =
    rec self : active_point .
        (x = px,
         y = py,
         d = λp:point . sqrt((p.x - self.x)**2 + (p.y - self.y)**2))
```

The second example illustrates the introduction of local storage in the definition of an object that closely resembles a typical instance in an object-oriented language:

```
type counter = (increment: int → int, fetch: unit → int)
val make_counter(n:int) =
    let count = cell n
    in (increment = λn:int . count := (get count) + 1,
        fetch = λnil:unit . get count)
```

Though these examples are very suggestive, Cardelli makes little comment on them.

In conclusion, Cardelli's semantics of inheritance may be summarized by the following principle: Inheritance *is* record subtyping.

The identification of inheritance with subtyping, proposed by Cardelli, does not explain inheritance in object-oriented languages. The analysis of Cardelli's proposal is complicated by the close relationship between inheritance and subtyping which was developed in Chapter 4, which used Cardelli's type system. The subtype relationship between parent and child, which is the focus of Cardelli's work, is a necessary effect and consequence of the inheritance mechanism presented in this thesis. But modelling one of its effects, no matter how significant, is not sufficient to explain a mechanism itself.

The *types* which are related by Cardelli's subtype scheme are not analogous to the *classes* in object-oriented programming. A record type can only define the format of instances, specifying the names and parameter types of functions which might represent methods. A *class* is a mechanism for constructing objects and providing them with similar behavior. Though a type can describe the instances of a class, it is not a class itself. The class concept is more closely modelled in type theories by the *elements* of existential types, again illustrating the mismatched levels of classes and types. The denotational model presented in Section 6, though abandoning claims to type-correctness, does more closely models classes.

There is also a structural mismatch between Cardelli's notion of subtyping and object-oriented inheritance. In object-oriented programming, inheritance is a *mechanism*, not a *relation* as it is in Cardelli's model. Inheritance is not discovered about classes in an object-oriented program after it has been written, rather, inheritance is a means to build programs. The semantics of inheritance presented in Section 2.3 defines inheritance as a mechanism. This mechanism is shown to capture the essential aspects of object-oriented inheritance in Chapter 5.

Cardelli's proposal for explicit selection of methods at object-creation time is not general enough to model the selection of methods in object-oriented programming. This mechanism only works as long as the functions being used to implement methods do not depend upon other methods in the class. If they do, then it is impossible to choose them independently; they are inextricably part of a composite structure. In addition, the explicit selection of methods does not address the interaction between original methods and their replacements arising from the use of super. The selection of interacting methods and invocation of original methods via super are both handled by the semantics of record inheritance presented in Section 2.4.2.

In comparing the type definitions of car and vehicle with the the points and counter examples of explicit object creation, it is clear that point and counter more closely resemble object-oriented class definitions. Cardelli's model of inheritance is defined as a relationship between car and vehicle, while the semantics of inheritance presented in this thesis is based on inheritance of behavior from point or counter.

Others researchers have challenged Cardelli's identification of inheritance and subtyping. Snyder [25] argues that inheritance does not always correspond to subtyping, especially when methods may be deleted. Wegner [30] distinguishes between classes and types, and discusses several of their subtle interactions. Danforth and Tomlinson [9] discuss difficulties in an attempt to use Cardelli's semantics to reproduce the effect of inheritance in object-oriented languages.

11.2 Kamin

Kamin [14] presents a denotational semantics of inheritance in Smalltalk-80. In his semantics, inheritance is handled by building a structure containing all the class definitions in the system, and then taking a single global fixpoint to derive the meaning of the entire system in one operation. His semantics employs the traditional techniques of standard semantics, including continuations to express method returns and blocks.

Kamin defines a fairly complete subset of Smalltalk-80, including a modified form of metaclasses. He defines a Smalltalk-80 program as a mapping from class names to class definitions, which in turn are mappings from message keys to method expressions. The entire program is self-referential, because one class definition may inherit from another, and also because expressions may invoke class methods.

Kamin's semantics defines the meaning of a Smalltalk program by a single global fixpoint that resolves all class references in one stroke. Message sending, recursion, and inheritance are all handled by this single fixpoint construct. But in his conclusion, Kamin asks "Should these not be *distinct* and *separable* features?". In addition, the the necessity of a global fixpoint for inheritance leads Kamin to conclude that "inheritance is strictly global".

Kamin also introduces some syntactic changes to simplify the semantics. The semantics of the pseudo-variable super is not defined directly. Noting that the class to which super refers may be determined statically in Smalltalk-80 programs (it is the superclass of the class in which the expression appears), Kamin requires super m e to be translated into a special form self C::m e where C is the explicit name of the superclass.

Class methods, part of the meta-class concept, are defined as a special expressions that have the effect of evaluating the method with the receiver bound to a special null object, which represents the fact that instance variables are not bound within a class method.

Though Kamin's semantics is an accurate description of Smalltalk-80, it is not *compositional.* A semantics is compositional if the meaning of a compound expression depends only upon the meaning of its component parts. Compositionality is one of the most desirable aspects of denotational semantics; it leads to modularity of definition, and indicates where equivalent expressions can be substituted because they have the same meaning. Kamin's semantics is non-compositional because the meaning of a class is not built up from the meanings of its components (its parent together with its local definitions) but is instead the result of a global fixpoint over all classes. It is impossible to identify the meaning of any individual class independent from the whole. As a result Kamin's semantics provides little help in determining when two class definitions are equivalent (for example, when one involves inheritance and the other does not). The semantics of inheritance presented in this thesis is more compositional, because every class is assigned an independent meaning, and the meaning of a subclass depends only upon the meaning it inherits.

Kamin's conclusion that inheritance is strictly global has as its basis a failure to distinguish the two forms of reference to class denotations: inheritance and instantiation. Since inheritance requires access to the generator of recursive behavior, while instantiation must use the behavior fixpoint, the two references have different requirements. In adhering to the traditional association of class identifiers with behaviors (instead of behavioral generators), Kamin is forced to use a global fixpoint over a single generator representing the all the class generators for the entire object-oriented library. The denotational semantics of Chapter 6, in introducing generators as the denotations of class identifiers, allows composition of generators to model inheritance, while the fixpoints of the generators can be taken as needed for instantiation.

Kamin's as blurs scope issues and inheritance. In Smalltalk, class names have global scope, so it is necessary for a global fixpoint to resolve references to class names. This use of the fixpoint is associated *only* with the binding of class denotations to identifiers, in the form of a global recursive binding, and is not associated with inheritance. On the other hand, recursion via self within a class definition refers only to the class, not to the entire program; thus a local fixpoint is appropriate for classes.

Because Kamin uses a single global fixpoint to resolve recursion and and inheritance, the relation between these concepts is not made clear. The semantics of inheritance defined in Section 2.3 distinguishes inheritance from recursion, at the same time indicating that they are definitely not separable.

Kamin's semantics defines a non-error value for programs that are *not* valid in Smalltalk-80. These implicit extensions to Smalltalk are caused primarily by the merging of classes and metaclasses which occurs in his semantics:

- The introduction of the form self C::m e allows classes to send messages directly to any of their ancestors, which is not allowed in Smalltalk-80. In its most general form, e C::m e allows any method to be invoked on an expression e as if it where defined on that expression. This is not allowed in Smalltalk-80 since it is a direct abrogation of object-oriented philosophy.
- Any method may erroneously invoke a method from any other class, and if the other class method does not use any instance variables not defined in the first class, then the method may produce a non-error value. In addition, class variables may be accessed globally. This oversight is perhaps excusable, because syntactic conditions are easily imposed upon programs to eliminate these problems.
- Class methods cannot be distinguished from instance methods by syntactic analysis. In effect, class methods are only distinguished from instance methods by the absence of instance variable references. It is possible to invoke a class method by a normal message-send to an instance with the the same effect as invoking the class method using the special class method expression. And a method may be used meaningfully as both a class method and an instance method, because a runtime check may be used to decide whether or not instance variables are available.

The semantics of Smalltalk presented in Section 8 separates classes and metaclasses, and provides independent but parallel inheritance at both levels.

Finally, Kamin's semantics of inheritance in focusing on Smalltalk does not address the problem of the interaction between inheritance and typing, perhaps because inheritance is examined only in Smalltalk, a weakly-typed language. The semantics of inheritance presented in this thesis is amenable to type analysis, as carried out in Chapter 4.

11.3 Reddy

Reddy [23] independently developed a denotational semantics of inheritance that is essentially the same as the one presented here. He sketched the denotational semantics of a series of languages dealing with objects, classes, and, finally, inheritance, which model the behavior of Smalltalk. His semantics uses local fixpoints, and recognizes the interaction between inheritance and recursion. However, his semantics is only applied to Smalltalk. In addition, he does not attempt to prove the correctness of his semantics, nor does he investigate applications of type theory.

11.4 Wolczko

Wolczko [32] described inheritance by a transformation from object-oriented programs with inheritance to programs in a simpler language without inheritance. The semantics of this simpler language was defined using denotational semantics. The abstract syntax of object-oriented programs with inheritance *Program'*, is translated into the syntactic domain of programs without inheritance, *Program*, by the mapping *PProgram*:

$PProgram : Program' \rightarrow Program$

The meaning of programs in *Program*, on the other hand, is mapped into the domain of program denotations, *Program_den*, by a valuation function:

$MProgram : Program \rightarrow Program\_den$

Wolczko used these techniques for a semantics of Smalltalk[31].

Though Wolczko provides a denotational semantics of class-based languages [29], his semantics of inheritance is essentially syntactic, not denotational. This can be seen in the use of a translation from programs into programs in the function *PProgram*, instead of a denotational mechanism for constructing the meaning of an inheritor from the meaning of its parent, as in this thesis.

11.5 Jorgensen

Jorgensen [22] presents an action semantics, a variety of denotational semantics, of inheritance in Beta. The full generality of Betais not examined, however, due to the following restriction on the use of virtual patterns (redefinable attributes):

> *"Virtual patterns can only be referred to in the method of the pattern [imperative] in which [they are] declared." [22, Page 16]*

This restriction essentially means that patterns cannot make self-references to their virtual (redefinable) components.

Chapter 12

Conclusion

Inheritance is explained as a mechanism for incremental programming. Its key aspect is the modification of self-reference in inherited components to refer to the derived result. This manipulation of self-reference allows inheritance to simulate the effect of destructive modification.

A denotational model of inheritance is developed, based on a traditional analysis of recursion. The model shows how inheritance exploits previously untapped potential in the use of fixpoints for the construction of programs. The essential innovation is the incremental construction of the arguments for the fixpoint function. A distributive algebra useful for expressing this construction is introduced to facilitate the definition of operations on generators. Inheritance is especially relevant to the modification of records, or collections of labeled attributes, because they are multi-faceted and a modification can be defined by a collection of attributes to add or replace. A variety of inheritance mechanisms are examined, including a generalization of inheritance that allows more flexible derivation than has previously been provided.

Several examples of novel uses of inheritance are given. These examples demonstrate that inheritance has wider applicability than simply object-oriented class definition.

To demonstrate the correctness of the model, it is proven to correspond directly to the operational implementations of inheritance in object-oriented languages.

The constraints induced by self-reference and inheritance are investigated using type-theory, resulting in a formal characterization of abstract classes and connections between inheritance and subtyping. Abstract classes in object-oriented programming are those which send messages to themselves that are not implemented; their generators don't satisfy the conditions on the fixpoint operator. The well-known effect of inheritance that children resemble their parent is a consequence of the change of self-reference within the parent to the child. But the minimal constraint is not that children are subtypes of their parents, but that they are subtypes of their parents self-reference.

The model is incorporated into standard denotational semantics for the analysis of object-oriented languages. The fundamental innovation is the addition of explicit generators, and adding constructs for their manipulation.

The semantics of inheritance in several object-oriented languages is then investigated, using a common framework in their inheritance mechanisms are easily compared. One significant discovery was that Smalltalk and Beta have the same basic inheritance mechanism, but it operates in a different direction in each language; a child in Smalltalk corresponds to a prefix in Beta.

Bibliography

[1] M. P. Atkinson and P. Buneman. Database programming languages. Technical Report Technial Report, University of Glasgow, 1975.

[2] R. J. R. Back and H. Mannila. A semantic approach to program modularity. *Information and Control*, 60:138–167, 1984.

[3] P. G. Bassett. Frame-based software engineering. *IEEE Transactions on Software Engineering*, pages 9–16, 1987.

[4] A. H. Borning and D. H. Ingalls. A type declaration and inference system for Smalltalk. In *Proc. of Conf. on Principles of Programming Languages*, pages 133–141, 1982.

[5] A. H. Borning and T. O'Shea. Deltatalk: An empirically and aesthetically motivated simplification of the Smalltalk-80 language. In *European Conference on Object-Oriented Programming*, pages 1–10, 1987.

[6] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, Lecture Notes in Computer Science 173*, pages 51–68. Springer-Verlag, 1984.

[7] L. Cardelli and P. Wegner. On understanding types data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[8] O.-J. Dahl and K. Nygaard. *The SIMULA 67 Common Base Language.* 1970.

[9] S. Danforth and C. Tomlinson. Inheritance and type theory. Technical report, MCC, 1987.

[10] J. Donahue. Object-oriented programming in Mesa (lecture notes). Workshop on Encapsulation, Modularity, and Reusibility, 1987.

[11] A. Goldberg and D. Robson. *Smalltalk-80: the Language and Its Implementation.* Addison-Wesley, 1983.

[12] M. J. C. Gordon. *The Denotational Description of Programming Languages.* Springer-Verlag, 1979.

[13] C. A. R. Hoare. Comments on Tennent's 'Denotational semantics of Hoare classes'. Unpublished correspondence, 1982.

[14] S. Kamin. Inheritance in Smalltalk-80: A denotational definition. In *Proc. of Conf. on Principles of Programming Languages*, pages 80–87, 1988.

[15] B. B. Kristensen, O. L. Madsen, B. Moller-Pedersen, and K. Nygaard. The Beta programming language. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 7–48. MIT Press, 1987.

[16] B. B. Kristensen, O. L. Madsen, B. Moller-Pendersen, and K. Nygaard. Abstraction mechanisms in the Beta programming language. *Information and Control*, 1983.

[17] H. Lieberman. Using prototypical objects to implement shared behavior in object-orietned systems. In *Proc. of Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 214–223, 1986.

[18] R. Milne and C. Strachey. *A Theory of Programming Language Semantics.* Chapman and Hall, 1976.

[19] J. C. Mitchell. Coercion and type inference (summary). In *POPL*, 1984.

[20] D. Moon. Object-oriented programming with flavors. In *Proc. of Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 1–9, 1986.

[21] P. Mosses and G. Plotkin. On proving limiting completeness. *SIAM Journal of Computing*, 16:179–194, 1987.

[22] J. Palsberg. An action semantics for inheritance. Master's thesis, University of Aarhus, 1988.

[23] U. Reddy. Objects as closures: Abstract semantics of object-oriented languages. In *Proc. ACM conf. Lisp and Functional Programming*, pages 289–297, 1988.

[24] D. S. Scott. Data types as lattices. *SIAM Journal*, 5(3):522–586, 1976.

[25] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proc. of Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 38–45, 1986.

[26] M. Stefik and D. G. Bobrow. Object-oriented programming: Themes and variations. *The AI Magazine*, 6(4):40–62, 1986.

[27] J. Stoy. *Denotational Semantics: The Scott-Strachy Approach to Programming Language Semantics.* MIT Press, 1977.

[28] R. D. Tennent. Denotational semantics of Hoare classes. Unpublished manuscript, 1982.

[29] P. Wegner. Dimensions of object-based language design. In *Proc. of Conf. on Object-Oriented Programming: Systems, Languages and Applications*, pages 168–182, 1987.

[30] P. Wegner. Inheritance as a mechanism for incremental modification. In *European Conference on Object-Oriented Programming*, 1988.

[31] M. Wolczko. Semantics of smalltalk-80. In *European Conference on Object-Oriented Programming*, pages 108–120, 1987.

[32] M. Wolczko. *Semantics of Object-Oriented Languages.* PhD thesis, University of Manchester, Deptartment of Computer Science, 1988.