Compiling Inheritance using Partial Evaluation *

Siau Cheng Khoo R.S. Sundaresh
Yale University
Department of Computer Science
New Haven, CT 06520
{khoo,sundaresh}@cs.yale.edu

Abstract

[CP89] presents two semantics – one denotational and one operational – for *inheritance*: A central concept of object oriented programming. We show how to use these semantics to *interpret* and *compile* inheritance. The main result is the elimination of compile time method lookups within an object instance. More specifically, this eliminates the inheritance references within an object. To the best of our knowledge, this is the first work on compiling inheritance based on a formal semantics.

We first demonstrate the necessity of converting the semantics into continuation passing style; then we look into the result of performing partial evaluation on their corresponding interpreters. Based on the *Futamura Projections*, we have also generated compilers from each of the interpreters.

1 Introduction

Inheritance, a central concept of object oriented programming, has been given a denotational semantics in [CP89]. By showing the equivalence of this denotational semantics with the widely accepted operational semantics, Cook and Palsberg [CP89] justify the claim that it accurately models inheritance. The benefits of inheritance as a method to gracefully extend programs are well known. Indeed, Cook and Palsberg identify the essence of inheritance as "a mechanism for deriving modified versions of recursive structures".

The benefits of inheritance come with a price: a naive implementation will be unacceptably slow because the meaning of a message can only be determined by traversing the class hierarchy (possibly many times). Most compilers for object oriented languages attempt to eliminate this overhead by trying to determine the method to be executed corresponding to a message at compile time. In this paper, we use the semantics of inheritance given in [CP89] in a prescriptive role by compiling inheritance based on them. As such, this work can be seen as fitting in the overall framework of semantics-directed compilation, an area which has been the subject of intense activity (e.g. [Mos76, Jon80, CK90, CD91b]). To the best of our knowledge, this is the first work on compiling inheritance based on a formal semantics.

A non-trivial semantic-directed compilation aims at eliminating as many static operations as possible. Since the semantic definitions – both operational and denotational – given for inheritance are developed primarily for explanatory purposes, they lack the distinction between static and dynamic semantics. Traditionally, such a lack of distinction forces one to determine the static semantics by hand. Besides the fact that their soundness are generally difficult to prove, this process is also error prone [Jon80, Ple87].

Partial evaluation offers a unified approach towards compiling and generating compiler from interpreters. It uses a *binding time analysis* to determine automatically the static semantics, and a *specializer* to execute the static operations so detected. This results in a robust and simple approach to semantic-directed compilation. A detailed description of such approach can be seen in [CD91b].

Frequently, how much of the static semantics can be detected depends on the way the semantics is written. More specifically, with a semantic definition written in direct style, it is possible that some static operations cannot be detected by the binding time analysis, and thus their corresponding expressions are made residual at compile time. This is further explained in Section 3.2. Such a shortcoming can be circumvented by modifying the semantic definition in a systematic manner so that more static operations can be detected. In particular, experience shows that better binding time information can be achieved by converting the definition into continuation passing style (CPS). This is investigated in detail by Consel and Danvy in [CD91a].

In this paper, we achieve the compilation of inher-

*This work was supported in part by NSF grant CCR-8809919. The first author was supported by a National University of Singapore overseas graduate scholarship. The second author was supported by an IBM graduate fellowship.

itance by first converting the semantic definition into one in continuation passing style (such conversion algorithms are already available, for example, in [DF90]), and then partially evaluating the corresponding interpreter with respect to some object instances. The primary result of the partial evaluation is the elimination of the *compile time method lookups.* Traditionally, such optimization is incorporated in the object oriented compilers (e.g. [CU89]).

This experiment is performed using Schism, a partial evaluator for a side-effect free dialect of Scheme [Con90a, Con90b]. The source programs are written in pure Scheme: a dynamically typed, applicative order implementation of lambda-calculus. Schism handles *higher order functions* as well as the *data structures* manipulated by the source programs, even when they are only *partially known.* The specializer of Schism is written in pure Scheme and is *self-applicable.* It can generate a compiler out of the interpretive specification of a programming language.

To summarize the results of the paper:

- Programs using inheritance are compiled into programs without inheritance. In other words, the decision of which method to execute in response to a message is done statically. The compiled program only needs to perform method dispatch in a single step.
- This process is provably correct since it is based on two automatic transformations of a formal semantics: CPS conversion and partial evaluation.
- The programs thus compiled typically show sizable speedups over their interpreted counterparts.

The paper is organized as follows: Section 2 describes *method systems*, the source language we will be compiling. The next two sections describe the issues involved in using the semantics for purposes of compilation. The overall strategy for compilation is as follows: we first derive interpreters from both the denotational and operational semantics. Compilation can be achieved by specializing these interpreters with a program (The first *Futamura projection* [Fut71]). Compiler generation is achieved using the second Futamura projection. We present the results of compilation using an example. In particular, Section 3 derives an interpreter from the operational semantics of inheritance, and Section 4 derives an interpreter from the denotational semantics.

2 Method Systems

Cook and Palsberg [CP89] introduce a simple formalization of object oriented languages called *method systems.* The aim of this language is to encompass only those language features directly related to inheritance. Other important features of object oriented languages

Syntactic Domains :

ρ	$\in$	*Instance*	Instances
κ	$\in$	*Class*	Classes
m	$\in$	*Key*	Messages
f	$\in$	*Primitive*	Primitives
e	$\in$	*Exp*	Methods

$e \;:=\; Self \mid Super \mid arg \mid n \mid e_1\ m\ e_2 \mid f\ (e_1, \ldots, e_n)$

Syntactic Operations :

$class : Instance \rightarrow Class$
$parent : Class \rightarrow (Class + Error)$
$methods : Class \rightarrow Key \rightarrow (Exp + Error)$

Figure 1: Syntactic Domains

```
(defineType Classes
  (Error)
  (Root)
  (Clss methods
        superclass))

(defineType Expr
  (Error-exp)
  (Exp expression))
```

```
(defineType Exp-type
  (Self)
  (Super)
  (Arg)
  (Sending e1 m e2)
  (Number n)
  (Var x)
  (Prim-+ e1 e2)
  (Prim-- e1 e2)
  (Prim-* e1 e2)
  (Prim-< e1 e2)
  (Prim-sqrt e1)
  (Prim-max e1 e2))
```

Figure 2: Abstract Syntax

like instance variables, assignment and object creation are omitted. Since we are interested in aspects of compilation related to inheritance, we will use this language for our experiments.

An *object* is an *instance* of a *class.* A class is a mapping from *messages* to *expressions* and may inherit methods from its parent. The relation between these syntactic domains can be seen in Figure 1. Also shown are useful operations on these domains. Self reference is denoted by *Self* and reference to the parent by *Super.* The argument to the expression is referenced via *arg.* Message passing is denoted by $e_1\ m\ e_2$. This sends the message m with argument e_2 to the object e_1. Note that objects are fully first class. They can be passed as arguments and returned as results.

The operation *class* returns the class to which an instance belongs. *parent* returns the parent of a class, returns an error if applied to the root. *methods* returns the method expression when applied to a class and a message. It returns error if the message is undefined in the class.

In Schism these domains are encoded using Schism type constructs. These constructs are simplified version of ML constructs. More specifically, the construct `defineType` defines a product or a sum, depending on whether it contains single clause or more. The constructs `let` and `let*` perform destructuring operations on elements of products besides creating new bindings.

Class Point

```
class point (a,b)
  method x = a
  method y = b
  method distFromOrig =
         sqrt(self.x * self.x
              + self.y * self.y)
  method closerToOrig(p) =
         self.distFromOrig < p.distFromOrig
```

Class Circle

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

Figure 3: Definitions of Class Circle and Point

Destructuring an element of a sum is obtained via the construct `caseType`, which is a conditional on the injection tag of the components of a sum. The encoding of the types is shown in Figure 2. The syntactic operations are coded as functions on these types (in Appendix A).

Figure 3 shows an example program in this language. The program implements two classes: a circle class, and a point class. The circle class inherits methods from the point class. Note the use of self reference in the method corresponding to `closerToOrig` in the class `point`. In a circle instance, this self reference will invoke the `distFromOrig` method of the `circle` class, not the `point` class. We will use this example later on to illustrate the results of the compilation process.

3 Operational Semantics and Its Partial Evaluation

In this section, we focus on the operational semantics of the inheritance. Although the techniques used here are illustrated on the operational semantics, they are equally applicable to the denotational semantics, as we shall see in Section 4. We first look at the problem incurred by using the operational semantics as described in [CP89] in a prescriptive role. We then solve the problem via systematic transformation of the semantics into continuation passing style. Next, we obtain an interpreter by direct encoding of the transformed semantics, and describe the result of partial evaluation through some examples.

The operational semantics of inheritance described in [CP89] models its method lookup mechanism, and is shown in Figure 4. The behavior of an instance is defined to be a mapping from messages to value transformers. The behavior function thus has a mapping for every message the instance responds to. Note that objects are first class: They can be passed as arguments and returned as results of computations. In particular, expression *self* refers to the behavior of the instance under consideration and expression *super* refers to the behavior of the parent of the class under consideration. Together, *self* and *super* capture the notion of inheritance in the sense that, when they are passed a message, they refer to either the instance itself or its parent to obtain the corresponding method for execution.

The meaning of the model is that of function *send*, which takes as arguments an object instance, a message key and a message argument. The latter two arguments are used to activate computation. In the context of compilation, it is natural that the instance be known, but both the message key and message argument be unknown. Similar static information is provided at the partial evaluation. Since both the constructs *self* and *super* refer to known instances, any known messages sent to them can be resolved by performing static method lookup. Although the message key and message argument input to the model are unknown, static messages exist in method expressions defined in the class. For instance, in the method `distFromOrig` defined in point class (Figure 3), all messages sent to *self* are static. Therefore, *our approach to compilation is to eliminate inheritance*; *i.e*, to eliminate any static messages sent to both *self* and *super*, so that the compiled program can perform method dispatch in a single step, regardless of the class in which the method was defined.

3.1 The Problem

Notice that the meaning of a method expression, including that of *self* and *super*, is defined in the valuation function *do*. In order to perform static method lookup, the first three arguments of function *do* must be static. More specifically,

1. *Exp* argument must be static so that the syntactic constructs (*self*, *super* and messages passed to them) can be identified;
2. *Instance* argument must be static so that both *self* and *super* can refer to the right instance;
3. *Class* argument must be static so that *super* can refer to the parent class.

Given that, in the context of compilation, the main function of the semantic definition, *send*, is passed a static instance, let us now examine the result of propagating this binding time information to the other valuation functions. We observe that:

1. Both the arguments *class* and *instance* in all the valuation functions are known at compile time.
2. Function *methods* returns a dynamic value at compile time.

1. Syntactic Domains:

ρ	$\in$	*Instance*	Instances
c	$\in$	*Class*	Classes
m	$\in$	*Key*	Messages
f	$\in$	*Primitive*	Primitives
e	$\in$	*Exp*	Methods

$e \;::=\; Self \mid Super \mid arg \mid n \mid e_1\ m\ e_2 \mid f\ (e_1, \ldots, e_n)$

2. Semantics Domains :

n	$\in$	*Number*	
α	$\in$	*Value*	$= Behavior + Number$
σ, π	$\in$	*Behavior*	$= Key \rightarrow (Fun + Error)$
ϕ	$\in$	*Fun*	$= Value \rightarrow Value$

3. Semantics Functions :

(a) Root Function :

$root : Class \rightarrow Boolean$

$root\ c = case\ (parent\ c)\ of$

$c' \in Class \rightarrow False$

$v \in Error \rightarrow True$

(b) Valuation Functions :

$send : Instance \rightarrow Behavior$

$send\ \rho = lookup\ (class\ \rho)\ \rho$

$lookup : Class \rightarrow Instance \rightarrow Behavior$

$lookup\ c\ \rho = \lambda\ m \in Key\ .\ case\ (methods\ c\ m)\ of$

$e \in Exp \rightarrow do\ [\![e]\!]\ \rho\ c$

$v \in Error \rightarrow (root\ c) \rightarrow Error\ [\!]$

$lookup\ (parent\ c)\ \rho\ m$

$do : Exp \rightarrow Instance \rightarrow Class \rightarrow Fun$

$do\ [\![self]\!]\ \rho\ c = \lambda\ \alpha\ .\ send\ \rho$

$do\ [\![super]\!]\ \rho\ c = \lambda\ \alpha\ .\ lookup\ (parent\ c)\ \rho$

$do\ [\![arg]\!]\ \rho\ c = \lambda\ \alpha\ .\ \alpha$

$do\ [\![e_1\ m\ e_2]\!]\ \rho\ c = \lambda\ \alpha\ .\ (do\ [\![e_1]\!]\ \rho\ c\ \alpha)m\ (do\ [\![e_2]\!]\ \rho\ c\ \alpha)$

$do\ [\![f\ (e_1, \ldots, e_n)]\!]\ \rho\ c = \lambda\ \alpha\ .\ f\ (do\ [\![e_1]\!]\ \rho\ c\ \alpha, \ldots, do\ [\![e_n]\!]\ \rho\ c\ \alpha)$

Figure 4: Operational Semantics for Inheritance

```
methods : Class → Key → (Exp + Error)
methods c m =
  case c of
    (d : ds) ∈ Class  →  lookup-methods (d : ds) m
    v ∈ Error         →  Error

lookup-methods : Mtds → Key → (Exp + Error)
lookup-methods nil m = Error
lookup-methods (d : ds) m =
  (mid d = m) → exp d ‖
                lookup-methods ds m
```

Figure 5: Function *methods* – direct style

The last observation comes from the fact that the value of the message key argument of the function *methods* (displayed in Figure 5[1]) is unknown at compile time. This means that it is not possible to determine which method within the object will be chosen. Therefore, the test in the case statement of function *lookup* is unknown and function *do* (one of the branches of the case statement) will thus have unknown first argument. Consequently, in the context of partial evaluation, the supposedly static method lookup cannot be detected, and none of them can be eliminated at compilation.

3.2 The Solution

We observe that the semantic definition of inheritance written in direct style does not provide as much static information as we expect. As a simple instance of the problem, consider a conditional expression whose test result is unknown but whose branches are static: The static values cannot be passed to the expression enclosing the conditional. This is exactly the case in the given operational semantics for inheritance, where function *methods* is unable to provide static information to its consumers: functions *do* and *lookup*.

This problem can be solved by converting the semantic definition into continuation passing style. Figure 6 displays functions *lookup* and *methods*, both in continuation passing style. Notice that the continuation argument for *methods* (*i.e.*, *Ke*) is passed static information (either an expression or an error message) at each branch of the case expression. As one such continuation is a call to function *do*, the latter function is passed a static *Exp* argument. Thus, the following property about the resulting semantic definition follows:

> The first three arguments of function *do* are known at compile time.

[1] As *methods* is a syntactic operation, its meaning is not defined in the operational semantics. Here, we provide its semantics with the assumption that a class contains a list of methods, denoted by (*d* : *ds*). Furthermore, for each method *d*, function *mid* retrieves the method name, while function *exp* retrieves the method expression.

```
lookup : Class → Instance → Key
         → Value → Fun → Value
lookup c ρ =
  λ m α Kv .
    method c m
      (λ e ∈ (Exp + Error) .
        case e of
          v ∈ Error → (root c) → Kv Error ‖
                      lookup (parent c) ρ m α Kv
          e ∈ Exp → do e ρ c α Kv)

methods : Class → Key
          → ((Exp + Error) → Value) → Value
methods c k Ke =
  case c of
    (d : ds) ∈ Class → lookup-methods (d : ds) m Ke
    v ∈ Error → Ke Error

lookup-methods : Mtds → Key →
                 ((Exp + Error) → Value) → Value
lookup-methods nil m Ke = Ke Error
lookup-methods (d : ds) m Ke =
  (mid d = m) → Ke (exp d) ‖
                lookup-methods ds m Ke
```

Figure 6: Functions *lookup* and *methods* – CPS style

Knowing these arguments implies that occurrences of both *self* and *super* can be determined, and static messages passed to these inheritance constructs can be eliminated at compile time. Thus, *inheritance is eliminated at compile time.*

3.3 The Interpreter

We obtain an interpreter for the model by a direct transliteration of the semantics written in continuation style. It is displayed in Figure 7. Due to space limitation, we use `prim-op` to represent binary operators `prim-+`, `prim--`, `prim-*`, `prim-<` and `prim-max`. This interpreter is used in partial evaluation to produce the compiled program.

3.4 Obtaining Polyvariant Binding Time Behavior

In the context of partial evaluation, compilation is accomplished by specializing the interpreter with respect to the instance argument. This implies that for the main function *S*, the instance argument is static, while the message key and the message argument are left dynamic.

As an offline partial evaluator, Schism relies on the *binding time analysis* [Con90a] to determine the static expressions in the interpreter, prior to specialization. Specifically, this analysis computes *binding time signatures* for each function in the program. This consists of

```
;main function
(define (S rho m alpha) (send rho m alpha kId))

(define (send rho m alpha Kv)
 (filter (if (and (stat? rho) (stat? m))
             UNFOLD SPECIALIZE)
         (list rho m alpha Kv))
  (lookup (class rho) rho m alpha Kv))

(define (lookup k rho m alpha Kv)
  (method k m
    (lambda (e)
      (caseType e
        ([Error-exp]
         (caseType k
           ([Error] 'Error)
           ([Root] 'Error)
           ([Clss - parent-k]
            (lookup parent-k rho m alpha Kv))))
        ([Exp e] (do' e rho k alpha Kv))))))

(define (do' e rho k alpha Kv)
  (filter (if (stat? e) UNFOLD SPECIALIZE)
          (list e rho k alpha Kv))
  (caseType e
    ([Self]
     (Kv (lambda (m alpha Kv')
           (send rho m alpha Kv'))))
    ([Super]
     (Kv (lambda (m alpha Kv')
           (lookup (parent k)
                   rho m alpha Kv'))))
    ([Arg] (Kv alpha))
    ([Number n] (Kv n))
    ([Var x] (Kv (lookup-var (car rho) x)))
    ([Sending e1 m e2]
     (do' e1 rho k alpha
       (lambda (v1)
         (do' e2 rho k alpha
              (lambda (v2) (v1 m v2 Kv))))))
    ([Prim-op e1 e2]
     (do' e1 rho k alpha
       (lambda (v1)
         (do' e2 rho k alpha
           (lambda (v2) (Kv (op v1 v2)))))))
    ([Prim-sqrt e1]
     (do' e1 rho k alpha
          (lambda (v1)
            (Kv (sqrt (->float v1))))))
    ))
```

Figure 7: Interpreter Obtained from Operational Semantics

the binding time information of all its arguments and that of its result. A function may have more than one possible binding time signature. In our case, a trace on the calling pattern of the interpreter reveals that function Send has two different binding time signatures[2]:

$$\begin{array}{lll} \textbf{send} & :: & \mathit{Instance} \rightarrow \mathit{Key} \rightarrow \mathit{Value} \rightarrow \mathit{Cont} \rightarrow \mathit{Value} \\ \textbf{send} & :: & \mathit{Stat} \rightarrow \mathit{Dyn} \rightarrow \mathit{Dyn} \rightarrow \mathit{Cls} \rightarrow \mathit{Dyn} \\ \textbf{send} & :: & \mathit{Stat} \rightarrow \mathit{Stat} \rightarrow \mathit{Dyn} \rightarrow \mathit{Cls} \rightarrow \mathit{Dyn} \end{array}$$

The first signature of send has dynamic value for its parameter *Key*. This is so because the key is propagated from the user input, which is unknown at partial evaluation time. On the other hand, a method expression of the form e_1 m e_2 provides the instance behavior of e_1 with a specific key, m (from the source program); this means the behavior is passed a static key.

In a *monovariant* binding time analysis, such as the one in Schism, the various binding time signatures of a function are folded into one signature. The folding produces a less accurate binding time signature for the function, and directly affects the quality of the residual program (since the actions taken by the specialization to unfold or specialize a function depends on the accuracy of the function's binding time signature). To avoid this approximation, code is duplicated for those functions that have multiple binding time signatures, and function calls are adjusted so that different versions of a function are called at different binding time contexts. In particular, the code for function do' is duplicated – and simplified – to make explicit the call points at which a behavior is known to be passed with static message key. This is shown in Appendix B.

Figure 8 displays the call graph of the final interpreter, where functions are duplicated once to capture different binding time contexts. The vertical dotted line divides the graph into two parts: the left subgraph contains functions that are passed dynamic message key, whereas the right subgraph contains functions with static message key. Function Do'' triggers a sequence of calls with this additional static information.

3.5 Results of Partial Evaluation

The upper code segment in Figure 9 shows the result of compiling away the inheritance when the interpreter is partially evaluated with respect to a circle instance whose class definition is given in Figure 3. Variables a, b and r are free in this program. They represent the instance variables of the circle class. Their values can be used to simplify the method expression. Indeed, this is actually done, as shown in the second code segment in the figure, with the appropriate values given to the parameters a, b and r. From the upper code segment,

[2]For brevity, we only describe three different binding time values in specifying the binding time signature of a function; we use *Stat* for *static*, *Dyn* for dynamic and *Cls* for functional information. Readers are referred to [Con90a] for further information on Schism's binding time analysis.

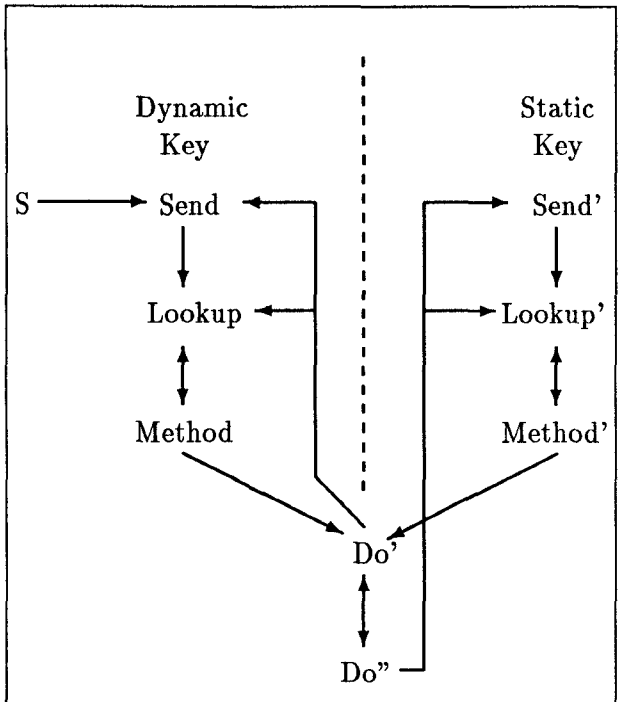

Figure 8: Call Graph for the Interpreter (Operational Semantics)

we notice that the function send has been specialized at partial evaluation, whereas the function lookup has been eliminated. Since the relationship between classes are available to the instance, Schism is able to compile away lookup calls to any class in the hierarchy (via *Self* and *Super*). For example, notice that all references to the circle instance in class Point (via expression *Self*) have been resolved. Therefore, methods in a class no longer refer to other classes; instead, dedicated codes are produced for different instances. Responding to a message now involves just a single lookup.

Figure 10 shows the result of partially evaluating the interpreter with respect to an instance that manipulates objects in a first-class manner (by passing them as arguments). We have also generated a compiler via self-application of Schism.

4 Denotational Semantics and Its Partial Evaluation

Figure 11 shows the Cook-Palsberg denotational semantics of method systems. An important point to note is that this semantics is compositional whereas the operational semantics is not. We will briefly explain the salient features of the semantics. The semantic object corresponding to an instance is a *Behavior* which is simply a mapping from messages to value transformers. The meaning of an instance is the fixpoint of the generator of the class to which the instance belongs. How

Skeleton of Residual Program with parameters a, b and r.

```
(define (s0 m alpha) (send1 m alpha kId))
(define (send1 m alpha kv)
  (cond ((eq? 'radius m) (kv 'r))
        ((eq? 'distFromOrig m)
         (kv (max (- (sqrt (+ (* a a) (* b b)))
                     'r)
                  '0)))
        ((eq? 'x m) (kv 'a))
        ((eq? 'y m) (kv 'b))
        ((eq? 'closerToOrig m)
         (alpha 'distFromOrig '-1
          (lambda (v4)
            (kv (< (max (- (sqrt (+ (* a a)
                                    (* b b)))
                           r) '0) v4)))))
        (else 'Error)))))
```

Residual Program with a = 3, b = 4 and r = 5.

```
(define (s0 m alpha) (send1 m alpha kId))
(define (send1 m alpha kv)
  (cond ((eq? 'radius m) (kv '5))
        ((eq? 'distFromOrig m) (kv '0))
        ((eq? 'x m) (kv '3))
        ((eq? 'y m) (kv '4))
        ((eq? 'closerToOrig m)
         (alpha 'distFromOrig '-1
          (lambda (v4) (kv (< '0 v4)))))
        (else 'Error)))))
```

Figure 9: Residual Program for a Circle Instance

Class Circle

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

Class Point

```
class point (a,b)
  method x    = a
  method y    = b
  method z    = self
  method w    = (self.z).x
  method u(p) = p.radius
  method v    = (self.u)(self)
  method distFromOrig = sqrt(self.x * self.x
                             + self.y * self.y)
```

Residual Program

```
(define (s0 m alpha) (send1 m alpha kId))
(define (send1 m alpha kv)
  (cond ((eq? 'radius m) (kv '5))
        ((eq? 'distFromOrig m) (kv '7))
        ((eq? 'x m) (kv '3))
        ((eq? 'y m) (kv '4))
        ((eq? 'z m) (kv send1))
        ((eq? 'w m) (send1 'x '1 kv))
        ((eq? 'u m) (alpha 'radius '1 kv))
        ((eq? 'v m) (send1 'radius '1 kv))
        (else 'Error)))
```

Figure 10: Class Definitions and Residual Program of Instance with first-class behavior (a= 3, b = 4, r = 5)

1. Semantic Domains :

$$
\begin{array}{rcll}
n & \in & Number & \\
\alpha & \in & Value & = Behavior + Number \\
r, \sigma, \pi & \in & Behavior & = Key \rightarrow (Fun + Error) \\
\phi & \in & Fun & = Value \rightarrow Value \\
\varrho & \in & Generator & = Behavior \rightarrow Behavior \\
\omega & \in & Wrapper & = Behavior \rightarrow Generator
\end{array}
$$

2. Semantics Functions :

(a) Root Function remains intact.

(b) Layer Operator :

$$\oplus : (Behavior \times Behavior) \rightarrow Behavior$$

$$r_1 \oplus r_2 = \lambda\, m \in Key\ .\ case\ (r_1\ m)\ of$$

$$\begin{array}{rcll} \phi & \in & Fun & \rightarrow \phi \\ v & \in & Error & \rightarrow r_2\ m \end{array}$$

(c) Inheritance Operator :

$$\boxed{\triangleright} : (Wrapper \times Generator) \rightarrow Generator$$

$$\omega \boxed{\triangleright} \varrho = \lambda\, self \in Behavior\ .\ (\omega\ self\ (\varrho\ self)) \oplus (\varrho\ self)$$

(d) Valuation Functions :

$$behave : Instance \rightarrow Behavior$$

$$behave\ \rho = fix\ (gen\ (class\ \rho))$$

$$gen : Class \rightarrow Generator$$

$$gen\ c = (root\ c) \rightarrow \lambda\, \sigma \in Behavior\ .\ \lambda\, m \in Key\ .\ Error$$

$$[\!]\ (wrap\ c) \boxed{\triangleright} (gen\ (parent\ c))$$

$$wrap : Class \rightarrow Wrapper$$

$$wrap\ c = \lambda\, \sigma\ .\ \lambda\, \pi\ .\ \lambda\, m \in Key\ .\ case\ (methods\ c\ m)\ of$$

$$\begin{array}{rcll} e & \in & Exp & \rightarrow eval\ [\![e]\!]\ \sigma\ \pi \\ v & \in & Error & \rightarrow Error \end{array}$$

$$eval : Exp \rightarrow Behavior \rightarrow Behavior \rightarrow Fun$$

$$eval\ [\![self]\!]\ \sigma\ \pi = \lambda\, \alpha\ .\ \sigma$$

$$eval\ [\![super]\!]\ \sigma\ \pi = \lambda\, \alpha\ .\ \pi$$

$$eval\ [\![arg]\!]\ \sigma\ \pi = \lambda\, \alpha\ .\ \alpha$$

$$eval\ [\![e_1\ m\ e_2]\!]\ \sigma\ \pi = \lambda\, \alpha\ .\ (eval\ [\![e_1]\!]\ \sigma\ \pi\ \alpha)m\ (eval\ [\![e_2]\!]\ \sigma\ \pi\ \alpha)$$

$$eval\ [\![f\ (e_1, \ldots, e_n)]\!]\ \sigma\ \pi = \lambda\, \alpha\ .\ f\ (eval\ [\![e_1]\!]\ \sigma\ \pi\ \alpha, \ldots, eval\ [\![e_n]\!]\ \sigma\ \pi\ \alpha)$$

Figure 11: Denotational Semantics for Inheritance

is this generator functional obtained? This is the key insight of the semantics. It can be obtained by combining the generator corresponding to the parent class and a *wrapper* obtained from the child class. A wrapper takes two arguments (*self* and *super*) and returns a behavior which responds to messages defined in the child class. The definition of the operator $\boxed{\triangleright}$ defines the way of combining the "meanings" of the child and the parent. It is interesting to note the delayed binding of *self* in the expression. This is crucial in obtaining the *late binding* effect of object oriented languages.

Just as the operational semantics, the denotational semantics is converted into continuation passing style so that inheritance can be made static.

4.1 Interpreter

The interpreter, in continuation passing style, obtained from the denotational semantics processes first-order method system programs (see appendix C). We plan to extend this to a higher-order language. The continuation in function `method` of the interpreter now gets a static argument, after dispatch is done in function `lookup-methods`. This ensures that function `eval'` receives static first argument, and we are once again able to eliminate the inheritance from the residual program after partial evaluation.

4.2 Binding Time Behavior

The main function of the interpreter, `behave`, takes three arguments: an instance, a message and an argument. Of these only the instance is static. The binding time analysis of Schism propagates this information to obtain binding time signatures for each function. As mentioned in the section on operational semantics, the binding time analysis of Schism is monovariant. Therefore, it introduces a loss of information when a function is called in two places with different binding times for the arguments. This happens in the case of the `beh` function. Initially it is called with the key argument *dynamic* but later calls (in `eval`) have a static key argument. This necessitates a duplication of the `beh` function, and also the functions which it refers to.

Figure 12 shows the call graph for the interpreter. The vertical dotted line separates the graph into two parts: one with the message key static and the other with the message key dynamic.

4.3 Results of Partial Evaluation

The result of specializing the interpreter on a simple method system program can be seen in Figure 13. It is interesting to note that all functions except `beh` have been unfolded. This means that all the method resolution has occurred at specialization time. The class structure has been collapsed into a single lookup. A

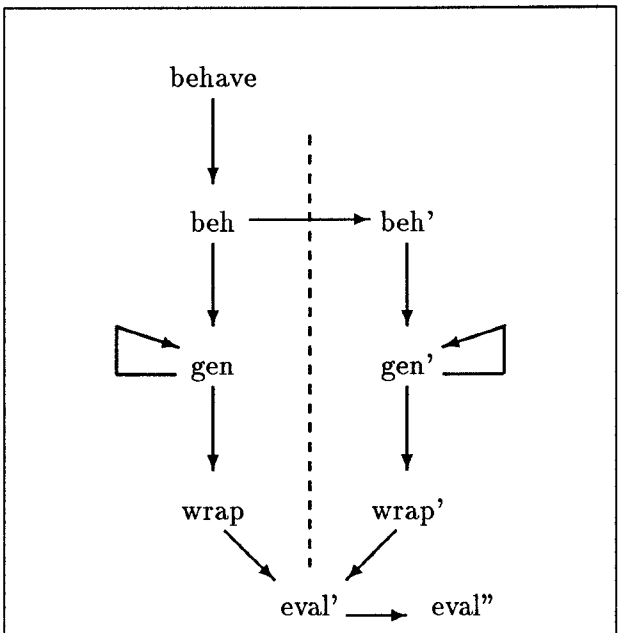

Figure 12: Call Graph for the Interpreter (Denotational Semantics)

Class Point

```
class point (a,b)
  method x = a
  method y = b
  method distFromOrig = sqrt(self.x * self.x
                             + self.y * self.y)
```

Class Circle

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

Skeleton of Residual Program with parameters a, b and r.

```
(define (behave0 m alpha)
  (cond ((eq? 'radius m) r)
        ((eq? 'distFromOrig m)
         (max (- (sqrt (+ (* a a) (* b b)))
                 r) 0))
        ((eq? 'x m) a)
        ((eq? 'y m) b)
        (else 'error)))
```

Figure 13: Example Method System and Its Residual Program

compiler has also been derived from the interpreter via self-application of the partial evaluator.

5 Conclusion

We used the operational and denotational semantics of inheritance to construct interpreters for a simple language with inheritance called *method systems*. We first systematically converted the semantics into one in continuation passing style. The corresponding interpreters have the property that inheritances (*i.e.*, occurrences of *self*, *super* and static messages passed to them) are known at compilation time. Then, using the Futamura projections, these interpreters were used to compile programs and generate compilers. The implementation is based on the partial evaluator Schism. The compilation produces programs where the method lookup associated with inheritance is removed. The resulting programs show impressive speedups over their interpreted counterparts. This technique could be used profitably to extend languages to include inheritance in a simple and effective manner.

An interesting extension to this framework is the compilation of multiple inheritance, where a class can have more than one parent. It should be possible to compile out the process of deciding which methods will be executed in response to a message.

Acknowledgments. Thanks to Charles Consel for making Schism available, answering many questions and providing very thoughtful comments on earlier versions of this paper.

References

[CD91a] C. Consel and O. Danvy. For a better support of static data flow. Technical Report CIS-91-3, Kansas State University, Manhattan, Kansas, USA, 1991.

[CD91b] C. Consel and O. Danvy. Static and dynamic semantics processing. In *ACM Symposium on Principles of Programming Languages*, 1991. Also Yale Research Report 761.

[CK90] C. Consel and S. C. Khoo. Semantics-directed generation of a prolog compiler. Research Report 781, Yale University, New Haven, Connecticut, USA, 1990.

[Con90a] C. Consel. Binding time analysis for higher order untyped functional languages. In *ACM Conference on Lisp and Functional Languages*, 1990.

[Con90b] C. Consel. *The Schism Manual.* Yale University, Department of Computer Science, November 1990.

[CP89] W. Cook and J. Palsberg. A denotational semantics of inheritance and its correctness. In *OOPSLA'89 Conference Proceedings*, volume 24 of *SIGPLAN Notices.* ACM Press, 1989.

[CU89] C. Chambers and D. Ungar. Customization: Optimizing compiler technology for self, a dynamically-typed object-oriented programming language. In *ACM SIGPLAN'89 Conference on Programming Language Design and Implementation*, volume 24 of *SIGPLAN Notices.* ACM Press, 1989.

[DF90] O. Danvy and A. Filinski. Abstracting control. In *ACM Conference on LISP and Functional Programming*, 1990.

[Fut71] Y. Futamura. Partial evaluation of computation process-an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5), 1971.

[Jon80] N. D. Jones, editor. *Semantics-Directed Compiler Generation*, volume 94 of *Lecture Notes in Computer Science.* Springer Verlag, 1980.

[Mos76] P. D. Mosses. *Compiler generation using denotational semantics*, volume 45 of *Lecture Notes in Computer Science*, pages 436–441. Springer Verlag, 1976.

[Ple87] U. Pleban. Semantics-directed compiler generation. In *ACM Symposium on Principles of Programming Languages*, 1987. Tutorial.

A Schism Coding of Syntactic Operations

```
;class :: Instance --> Class
(define (class instance) (cdr instance))

;parent :: Class --> Class + Error
(define (parent cls)
  (caseType cls
    ([Error] (Error))
    ([Root] (Error))
    ([Clss - superclass] superclass)))

;method :: Class --> Key --> Kont-e
;          --> Exp + Error
(define (method cls key Ke)
  (caseType cls
    ([Error] (Ke (Error-exp)))
    ([Root] (Ke (Error-exp)))
    ([Clss mtds -]
     (lookup-methods mtds key Ke))))

(define (lookup-methods env key Ke)
  (filter (if (stat? env) UNFOLD SPECIALIZE)
          (list env key Ke))
  (cond ((null? env) (Ke (Error-exp)))
        ((eq? (car (car env)) key)
         (Ke (cdr (car env))))
        (else
         (lookup-methods (cdr env) key Ke))))

(define (lookup-var env key)
  (filter (if (stat? env) UNFOLD SPECIALIZE)
          (list env key))
  (cond ((null? env) (Error-exp))
        ((eq? (car (car env)) key)
         (cdr (car env)))
        (else (lookup-var (cdr env) key))))
```

B Duplicating *Do* to Achieve Polyvariant Behavior

```
;do' :: Exp --> Instance --> Class --> Value
;       --> Kont-v --> Value
(define (do' e rho k alpha Kv)
  (filter (if (stat? e) UNFOLD SPECIALIZE)
          (list e rho k alpha Kv))
  (caseType e
    ([Self]
     (Kv (lambda (m alpha Kv')
           (send rho m alpha Kv'))))
    ([Super]
     (Kv (lambda (m alpha Kv')
           (lookup (parent k)
                   rho m alpha Kv'))))
    ([Arg] (Kv alpha))
    ([Number n] (Kv n))
    ([Var x] (Kv (lookup-var (car rho) x)))
    ([Sending e1 m e2]
     (do'' e1 rho k alpha
       (lambda (v1)
         (do' e2 rho k alpha
              (lambda (v2) (v1 m v2 Kv))))))
    ([Prim-+ e1 e2]
     (do' e1 rho k alpha
       (lambda (v1)
         (do' e2 rho k alpha
              (lambda (v2) (Kv (+ v1 v2)))))))
    ([Prim-- e1 e2]
     (do' e1 rho k alpha
       (lambda (v1)
         (do' e2 rho k alpha
              (lambda (v2) (Kv (- v1 v2)))))))
    ([Prim-* e1 e2]
     (do' e1 rho k alpha
       (lambda (v1)
         (do' e2 rho k alpha
              (lambda (v2) (Kv (* v1 v2)))))))
    ([Prim-< e1 e2]
     (do' e1 rho k alpha
       (lambda (v1)
         (do' e2 rho k alpha
              (lambda (v2) (Kv (< v1 v2)))))))
    ([Prim-sqrt e1]
     (do' e1 rho k alpha
          (lambda (v1)
            (Kv (sqrt (->float v1))))))
    ([Prim-max e1 e2]
     (do' e1 rho k alpha
       (lambda (v1)
         (do' e2 rho k alpha
              (lambda (v2) (Kv (max v1 v2)))))))
    ))

  (define (do'' e rho k alpha Kv)
    (filter (if (stat? e) UNFOLD SPECIALIZE)
            (list e rho k alpha Kv))
    (caseType e
        ([Self]
         (Kv (lambda (m alpha Kv')
               (send rho m alpha Kv'))))
        ([Super]
         (Kv (lambda (m alpha Kv')
               (lookup (parent k)
                       rho m alpha Kv'))))
        ([Arg] (Kv (lambda (m alpha' Kv')
                     (alpha m alpha' Kv'))))
        ([Var x]
         (Kv (lambda (m alpha Kv')
               ((lookup-var (car rho) x)
                m alpha Kv'))))
        ([Sending e1 m e2]
           (Kv (lambda (m' alpha' Kv')
               ((do'' e1 rho k alpha
                  (lambda (v1)
                    (v1 m (do' e2 rho k alpha kId)
                                  (lambda (x) x))))
                m' alpha' Kv'))))
        ))
```

C Interpreter Derived from the Denotational Semantics

```
;behave :: Instance --> Key --> Value --> Value
  (define (behave rho m alpha)
    (beh rho m alpha))

  (define (beh rho m alpha)
    (filter SPECIALIZE (list rho m alpha))
    (((gen (class rho))
      (lambda (m alpha) (beh' rho m alpha)))
     m alpha))
```

```
  (define (beh' rho m alpha)
    (filter (if (and (stat? rho) (stat? m))
                UNFOLD SPECIALIZE)
            (list rho m alpha))
    (((gen' (class rho))
      (lambda (m alpha) (beh' rho m alpha)))
     m alpha))

;gen :: Class --> (Behavior --> Behavior)
  (define (gen k)
    (caseType k
      ([Error]
       (lambda (beh) (lambda (m alpha) 'Error)))
      ([Root]
       (lambda (beh) (lambda (m alpha) 'Error)))
      ([Clss - pk] (sub (wrap k) (gen pk)))
      ))

;sub :: Wrapper --> Generator --> Generator
  (define (sub W P)
    (lambda (self)
      (layer ((W self) (P self))
       (P self))))

;wrap :: Class --> Wrapper
;i.e. Class --> Behavior --> Generator
  (define (wrap k)
    (lambda (sigma)
      (lambda (pi)
        (lambda (m alpha)
          (method k m
            (lambda (exp)
              (caseType exp
                ([Error-exp] 'Error)
                ([Exp e]
                 (eval' e sigma pi alpha)))))))))

;layer :: Behavior --> Behavior --> Key
;         --> Value --> Value
  (define (layer r1 r2)
    (lambda (m alpha)
      (let ((phi (r1 m alpha)))
        (if (eq? 'Error phi)
            (r2 m alpha)
            phi))
      ))

;eval' :: Exp --> Behavior --> Behavior
;         --> Value --> Value
  (define (eval' e sigma pi alpha)
    (filter (if (stat? e) UNFOLD SPECIALIZE)
            (list e sigma pi alpha))
    (caseType e
      ([Arg] alpha)
      ([Number n] n)
      ([Sending e1 m e2]
       ((eval'' e1 sigma pi alpha)
        m (eval' e2 sigma pi alpha)))
      ([Prim-+ e1 e2]
       (+ (eval' e1 sigma pi alpha)
          (eval' e2 sigma pi alpha)))
      ([Prim-- e1 e2]
       (- (eval' e1 sigma pi alpha)
          (eval' e2 sigma pi alpha)))
      ([Prim-* e1 e2]
       (* (eval' e1 sigma pi alpha)
          (eval' e2 sigma pi alpha)))
      ([Prim-< e1 e2]
       (< (eval' e1 sigma pi alpha)
          (eval' e2 sigma pi alpha)))
      ([Prim-sqrt e1]
       (sqrt (->float (eval' e1 sigma pi alpha))))
      ([Prim-max e1 e2]
       (max (eval' e1 sigma pi alpha)
            (eval' e2 sigma pi alpha)))
      ))

(define (eval'' e sigma pi alpha)
  (filter (if (stat? e) UNFOLD SPECIALIZE)
          (list e sigma pi alpha))
  (caseType e
      ([Self] sigma)
      ([Super] pi)
  ))
```