1994

A Semantic Framework for Understanding the Behavior of Modules and Classes in Programming Languages

Mahesh DODANI

Benjamin Kok Siew GAN
Singapore Management University, benjamingan@smu.edu.sg

A Semantic Framework for Understanding the Behavior of Modules and Classes in Programming Languages

Mahesh H. Dodani[a] and Kok Siew Gan[a]

[a] Department of Computer Science, University of Iowa, Iowa City, IA 52242, USA

Abstract

Recent trends in programming language design and implementation are aimed at integrating the two most important constructs to combat complexity: modules and classes. Both constructs provide encapsulation, a cornerstone of reliable programming. These constructs differ in their approach to building hierarchies: modules define visibility of components while classes define visibility of functionality.

How can modules and classes be effectively integrated within a simple programming language? This question captures the essence of developing semantics of these constructs to ensure that they interact in a meaningful way.

This paper develops a formal denotational semantic framework for understanding the interaction between classes and modules within programming languages. This semantic framework is developed incrementally; beginning with a base object oriented language with support for classes, objects, message passing and inheritance; and followed by extensions that support modules. These extensions consider both conventional static compile-time as well as dynamic run-time module constructs.

Keyword Codes: D.3.1; F.3.1; F.3.2; F.4.3

Keywords: Programming Languages, Formal Definitions and Theory; Theory of Computation, Specifying and Verifying and Reasoning about Programs; Semantics of Programming Languages; Formal Languages

1. Introduction

Constructs supporting encapsulation and hierarchical composition are essential to modern programming languages. The two most popular choices for such constructs are modules and classes. Modules comprise definitions of related programming items (including variables, functions/procedures, and types). The scope (or visibility) of an item is restricted to the module where it is defined. This scope can be extended into another module by defining an import relationship. Classes comprise the definition of a set of objects. Classes define the items (including variables, methods, and types) that implement the behavior of each object. Objects are run-time instantiation of classes. A class can inherit items from another defined class. This inheritance relationship allows reuse of class definitions.

In the 90s, several languages have attempted to integrate modules and classes. Are both constructs necessary in a programming language? This question is answered in a recent paper [Szy92] which argues for, and provides examples of, the need for both constructs. If a programming language supports both constructs then an important issue is whether modules interact with classes, objects, or both. This issue has major repercussions on the behavior of the constructs. On one extreme, modules are static compile-time constructs for encapsulating and organizing groups of classes. On the other extreme, modules are dynamic run-time constructs for encapsulating and organizing the behavior of objects. Examples of both extremes exist: Modular Smalltalk [Wir88], Modula-3 [Car88, Dona89, Nel91], and Ada9X [Ros92, Taft92] supports static modules, while Contracts [Helm90, Hol92] represent a dynamic module construct. These extremes of module behavior allow us to explore a range of interactions between modules and classes.

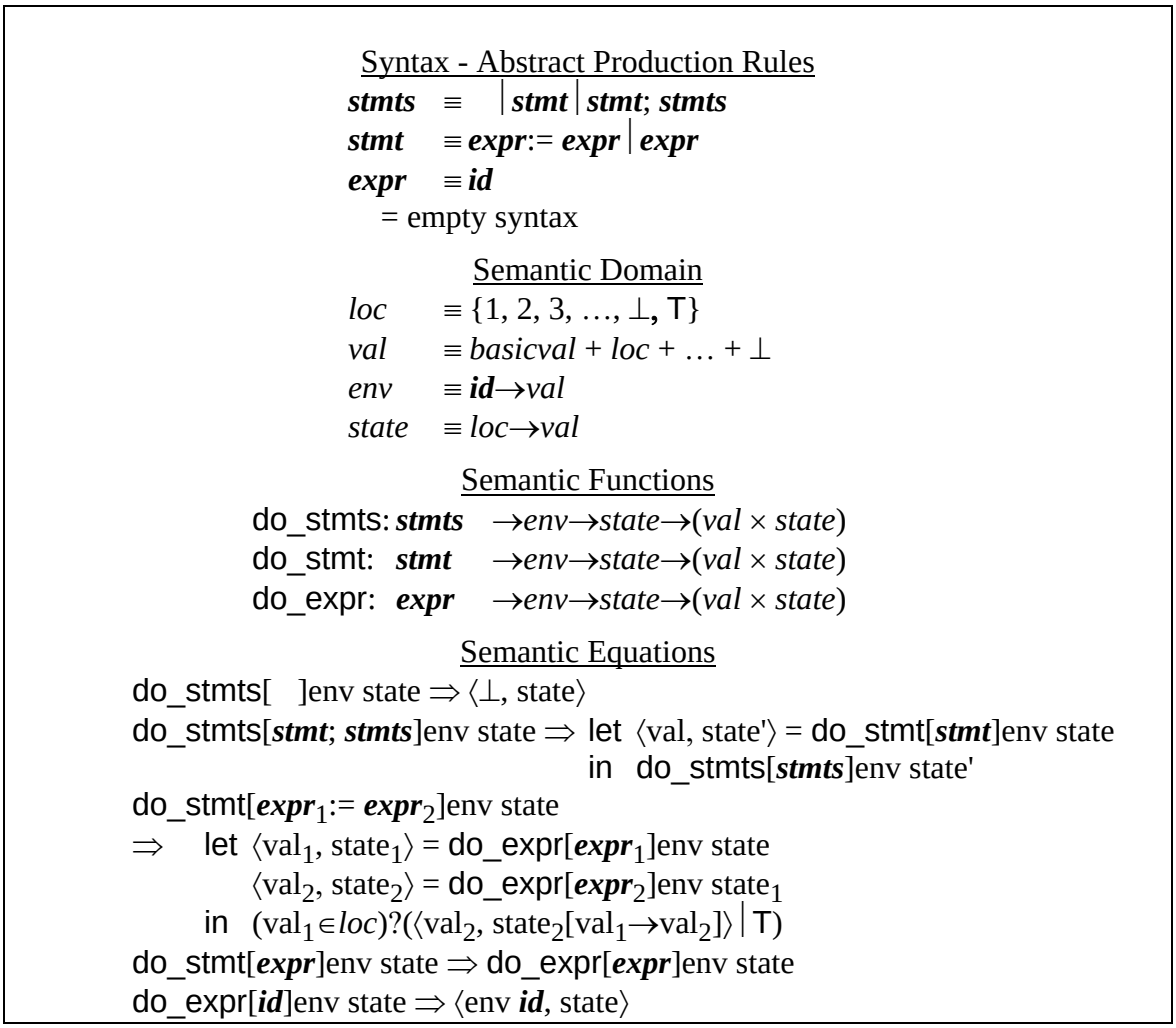
A precise understanding of the behavior of modules and classes within programming languages necessitates a semantic framework. This paper develops a denotational semantic framework to explore the two extreme behaviors of modules described above, and their interactions with classes in a programming language. The presentation is organized as follows: Section 2 develops the denotational semantics of a base Object-Oriented (OO) programming language that supports classes, objects, message passing, and inheritance. Section 3 extends the denotational semantics with support for static, compile-time module constructs. Section 4 extends the denotational semantics with support for dynamic, run-time module constructs which:

(1) provide encapsulation and define scope for objects as opposed to classes, and
(2) are instantiated and exist at run time.

The final section summarizes the presentation, analyzes the results, and presents future directions for research.

2. Denotational Semantics for Object-Oriented Language

This section develops denotational semantics for a typical object oriented language with support for classes, objects, message passing, and inheritance. The presentation is based on [Doda92, Kam88, Red88, Tsai92]. The following is a skeleton denotation of two basic expression statements:



Listing 1: Denotational semantic for expression.

The semantic domain *val* (value) denotes a *basicval* (basic value), a *loc* (location), or an ⊥ (undefined value). Later, new values will be added to *val* to represent structures like procedures, objects, classes, and dynamic modules. The *basicval* may be values in the following domains: integers, reals, characters, or boolean. Location *loc* represents an address in the underlying abstract machine. The domain *env* (environment) denotes the visibility scope for variables, and the domain *state* represents the memory of the abstract machine. The *state* takes a *loc* and maps it to its current *val*. T (top) represents an error, and ⊥ (bottom) represents undefined values. A conditional expression is represented by (condition)?(true expression | false expression). The semantics for the assignment statement (***expr***$_1$:= ***expr***$_2$), evaluates the expressions ***expr***$_1$ and ***expr***$_2$, checks the result of ***expr***$_1$, and binds the value of ***expr***$_2$ (val_2) to the location of ***expr***$_1$ (val_1). Note that an error occurs when val_1 is not a location. The expression ***id*** retrieves its value from the current environment.

The next step adds the syntax and semantics specific to OO concepts. These include support for defining a class, inheriting from a class by subclassing, sending a message to an object, and instantiating an object from a class.

Syntax - Abstract Production Rules

class ≡ CLASS ***id*** (***variabledecls***) ***classprocs*** END |
SUBCLASS ***id*** = ***typename*****$_{super}$** (***variabledecls***) ***classprocs*** END
variabledecls ≡ | ***variabledecl*** | ***variabledecl***; ***variabledecls***
variabledecl ≡ ***idlist***: ***typename*** | ***idlist***:= ***expr*** | ***idlist***: ***typename***:= ***expr***
classprocs ≡ | ***classproc*** | ***classproc classprocs***
classproc ≡ ***concreteproc*** | ***abstractproc***
concreteproc ≡ PROCEDURE ***id*** (***formals***):***typename*** = BEGIN ***stmts*** END
abstractproc ≡ ABSTRACT ***id*** (***formals***):***typename***
formals ≡ | ***formal*** | ***formal***; ***formals***
formal ≡ ***idlist***: ***typename***
expr ≡ ***id*****$_{object}$**.***id*****$_{mesg}$**(***exprs***) | NEW (***typename***)
typename ≡ ***id***
idlist ≡ ***id*** | ***id***, ***idlist***

Semantic Domain

val ≡ *basicval* + *loc* + *procval* + *classval* + *objval* + ⊥
procval ≡ (*state*→(*val*→)*((*val* × *state*) + ⊥))
penv ≡ ***id***→*procval*
clsdef ≡ *state*→(*env* × *state*)
clspenv ≡ *env*→*penv*
classval ≡ *clsdef* × *clspenv*
cenv ≡ ***id***→*classval*
objval ≡ *penv*

Semantic Functions

do_class:	***class***	→*env*→*cenv*
do_variabledecls:	***variabledecls***	→(*env* × *state*)→(*env* × *state*)
do_variabledecl:	***variabledecl***	→(*env* × *state*)→(*env* × *state*)
do_classprocs:	***classprocs***	→*env*→*penv*
do_classproc:	***classproc***	→*env*→*penv*
do_concreteproc:	***concreteproc***	→*env*→*penv*
do_abstractproc:	***abstractproc***	→*env*→*penv*
do_formals:	***formals***	→***idlist***
do_formal:	***formal***	→***idlist***
do_expr:	***expr***	→*env*→*state*→(*val* × *state*)
do_typename:	***typename***	→*env*→*val*

Listing 2: Denotational semantics specific to OO concepts.

The semantic domains *objval* and *penv* define an object as a mapping from ***id*** (procedure identifier) to *procval* (procedure value). Thus, procedure identifiers denote messages. Procedures are defined as functions that map a *state* and a list of finite actual arguments *val** into a new *val* and *state*. Note that an abstract procedure will result in ⊥ when called. The semantic domain *classval* (class value) is denoted by *clsdef* (class definition) and *clspenv* (class procedure environment). The *clsdef* takes a *state* and returns the *env* for local variables defined in the class,

along with a new *state* from the local declarations. The *clspenv* takes an *env* (the local declaration) and returns the procedure environment *penv*. Splitting *classval* into *clsdef* and *clspenv* enables the environments used by *clspenv* and *clsdef* to be defined at run time. This delay facilitates dynamic instantiation and late binding.

The semantic equations describing behavior of classes, inheritance and message passing are presented below. The semantics of a class definition is shown below:

```
do_class[CLASS id (variabledecls) classprocs END]env
⇒ let clsdef = λstate.do_variabledecls[variabledecls]⟨env, state⟩
      clspenv = λenv1.do_classprocs[classprocs](env;env1)
      classval = ⟨clsdef, clspenv⟩
  in [id→classval]
```

Listing 3: Semantic equation for CLASS.

The new classval contains clsdef and clspenv. The clsdef defines variables which are allocated space in the run time state. The clspenv defines class procedures which uses the environment (env_1) determined at run time. The environment (env_1) is retrieved from clsdef.

The semantics of inheritance is defined by the subclass construct shown below:

```
do_class[SUBCLASS id = typename_super (variabledecls) classprocs END]env
⇒ let ⟨clsdef_s, clspenv_s⟩ = do_typename[typename_super]env
      clsdef = λstate.  let ⟨env_s, state_s⟩ = clsdef_s state
                            ⟨env_1, state_1⟩ =
                                do_variabledecls[variabledecls]⟨env;env_s, state_s⟩
                        in  ⟨env_s;env_1, state_1⟩
      clspenv = λenv_2. let env_3 = env;env_2
                            penv_s = clspenv_s env_3
                            penv = do_classprocs[classprocs]env_3
                        in  penv_s;penv
      classval = ⟨clsdef, clspenv⟩
  in [id→classval]
```

Listing 4: Semantic equation for SUBCLASS.

This equation extends SUBCLASS with the superclass definition. First, the superclass (***typename*$_{super}$**) is retrieved from the environment. Second, clsdef appends the superclass environment (env_s) to its variable environment. Note that ";" represents the append operator. Third, clspenv includes the superclass procedure environment ($penv_s$). The environment (env_2) passed to clspenv denotes the variable environment defined in both the superclass and subclass.

Message passing semantics is defined as follows:

```
do_expr[id_object.id_mesg(exprs)]env state
⇒ let ⟨objval, state'⟩ = do_expr[id_object]env state
      expr_1,…,expr_n = exprs
      ∀i, 1≤i≤n, (⟨val_i, state_i⟩ = do_expr[expr_i]env((i≈1)?(state' | state_{i-1})))
      penv = objval
  in (penv id_mesg)state_n val_1…val_n
```

Listing 5: Semantic equation for message passing.

The identifier ***id*$_{object}$** is an objval from which a procedure environment (penv) is extracted. The procedure identifier ***id*$_{mesg}$** is used to identify the appropriate procedure *procval* in *penv*. Each ***expr***$_i$ is evaluated and passed as arguments ($val_1 \ldots val_n$) to the procval. Note that the state resulting from ***expr***$_{i-1}$ is used by ***expr***$_i$.

The clause NEW instantiates an object from a class. The semantics for object creation is defined as follows:

```
do_expr[NEW (typename)]env state
⇒ let classval = do_typename[typename]env
      ⟨clsdef, clspenv⟩ = classval
      ⟨env_1, state_1⟩ = clsdef state
      penvwithself = λpenv.clspenv(env_1[SELF→penv])
  in ⟨fix penvwithself, state_1⟩
```

Listing 6: Semantic equation for NEW.

The semantics for fix point is defined as follows:

```
fix: penv→penv ⇒ penv
fix penvwithself ⇒ let penv id = penvwithself penv id
                   in penv
```

Listing 7: Semantic equation for fix point.

The NEW clause dynamically instantiates an *objval* from a *classval* using the run time state. The instantiation produces an environment (env_1) from clsdef and a procedure environment (penv) from clspenv. SELF is recursively bound through the environment provided to clspenv, resulting in a *penv→penv* function (penvwithself). Taking the fix point of penvwithself results in an objval where all SELF references (including superclass definitions) are bound to the object. This late binding results from associating clspenv with (env_1[SELF→penv]) at runtime instead of at compile time when the CLASS is declared.

The semantic equations for do_variabledecl, do_classproc, do_concreteproc, do_abstractproc, do_formal, and do_typename have been omitted for brevity. Complete denotational specifications can be found in [Gan94].

3. Static Modules

This section, extends the denotational semantics of the OO language of section 2 with a static, compile time module construct. The module construct encapsulates a group of classes, thereby facilitating a local scope. Furthermore, the local scope allows definition of invariants among the classes in the module. Explicit import and export clauses facilitate a precise definition of visibility for the classes within the system.

The extension to the abstract syntax for static compile time modules is shown below:

compilations	≡	| ***compilation compilations***
compilation	≡	***interface*** | ***module***
interface	≡	INTERFACE ***id***; ***import decls*** END.
module	≡	MODULE ***id exports***; ***import decls*** END.
exports	≡	| EXPORT ***idlist***
import	≡	| IMPORT ***idlist***;
decls	≡	| ***decl decls***
decl	≡	***class***; | ***concreteproc***;

Listing 8: Syntax - Abstract Production Rules for static modules.

The program comprises a set of ***compilations***, which in turn are made up of ***module*** and/or ***interface***. Each ***module*** has a set of ***decls***. The environment used in ***decls*** are defined in the IMPORTed ***interface***. These interface definitions are replaced with EXPORTed definitions from the corresponding ***module*** definition.

The semantic domains for static compile time modules are the same as the semantic domains for OO languages as described in section 2. The semantic functions associated with static compile time modules are shown below:

execute_compilations:	***compilations***	$\rightarrow(val\rightarrow)^*(val \times state)$
do_compilations:	***compilations***	$\rightarrow(env \times state)\rightarrow(env \times state)$
do_compilation:	***compilation***	$\rightarrow(env \times state)\rightarrow(env \times state)$
do_interface:	***interface***	$\rightarrow(env \times state)\rightarrow(env \times state)$
do_module:	***module***	$\rightarrow(env \times state)\rightarrow(env \times state)$
do_exports:	***exports***	$\rightarrow env\rightarrow env\rightarrow env$
do_import:	***import***	$\rightarrow env\rightarrow env$
do_decls:	***decls***	$\rightarrow(env \times state)\rightarrow(env \times state)$
do_decl:	***decl***	$\rightarrow(env \times state)\rightarrow(env \times state)$

Listing 9: Semantic functions for static modules.

The semantics of executing a program is defined by execute_compilations which identifies the main procedure defined in the environment produced by do_compilation. This main procedure is executed with actual parameters $val_1 \ldots val_n$, resulting in a return value and new state. Note that the semantics ensure definition of variables before use.

```
execute_compilations[compilations] val1…valn
⇒ let ⟨env, state⟩ = do_compilations[compilations]⟨emptyenv, emptystate⟩
      procval = env id_main
  in  procval state val1…valn
```

Listing 10: Semantic equation for compilation units.

The semantics of handling computation units is defined by individual ***interface*** or ***module*** components. The semantics of an interface is shown below:

```
do_interface[INTERFACE id; import decls END.]⟨env, state⟩
⇒ let ⟨env1, state1⟩ = do_decls[decls]⟨do_import[import]env, state⟩
  in  ⟨env[id→env1], state1⟩
```

Listing 11: Semantic equation for INTERFACE.

Each ***interface*** consist of a name, imported environment and a declaration.

The semantics of modules is shown below:

```
do_module[MODULE id exports; import decls END.]⟨env, state⟩
⇒ let ⟨env1, state1⟩ = do_decls[decls](do_import[import]env) state
  in  ⟨do_exports[exports]env env1, state1⟩
```

Listing 12: Semantic equation for MODULE.

Each ***module*** produces a EXPORT environment and a new state resulting from the ***decls***. The environment used in ***decls*** is the IMPORTed environment.

The semantics defining the behavior of import and export relationships is shown below:

```
do_import[IMPORT    ;]env ⇒ emptyenv
do_import[IMPORT id;]env ⇒ [id→env id]
do_import[IMPORT id, idlist;]env
⇒ (do_import[IMPORT id;]env);(do_import[IMPORT idlist;]env)
do_exports[   ]env env1 ⇒ env
do_exports[EXPORT idlist]env env1
⇒ let id1,…,idn = idlist
  in  env[∀i, 1≤i≤n, (idi→
            let envi = env idi
            in  envi[∀j, (id'j∈domain(env1)∩domain(envi), id'j→env1 id'j)]
```

Listing 13: Semantic equations for IMPORT and EXPORT.

IMPORT definitions are used only in ***interface*** and ***module***. IMPORT returns the interface environment named in ***idlist***. These definitions must be exported by some module and have the effect of overriding the declaration with the same name in the interface.

4. Dynamic Modules

In stark contrast with the static compile time modules for encapsulating classes presented above, this section explores the semantics of a dynamic run time module that facilitates the definition and manipulation of relationships between objects. The dynamic nature of modules is similar to classes; that is, the scope is instantiated at run time. This run time instantiation is done explicitly using the RELATE clause (similar to NEW), and has the effect of a late binding between objects and parts in relationships. The binding itself is similar to parameter binding in methods: the actual objects are bound to formal parts of the dynamic module. Thus, modules are templates for defining relationships between objects. Each part of the template can be bound to an object when instantiated. The template can define shared variables, and prescribe local scoping rules.

This subsection extends the denotational semantics of the object-oriented programming language to support dynamic run time modules. Dynamic modules are defined using COMPOSE and SUBCOMPOSE clauses and instantiated at run time using the RELATE clause. These three clauses mirror CLASS, SUBCLASS and NEW defined for classes. RELATE binds run time objects to formal participants of the dynamic module defined by COMPOSE and SUBCOMPOSE. The resulting dynamic scope facilitates access to shared variables, local variables defined within participants, and actual objects bound to participants.

The following presents the abstract syntax for the language supporting the dynamic module:

decls	≡	| ***decl decls***
decl	≡	***compose***; | ***class***; | ***concreteproc***;
compose	≡	COMPOSE ***id*** (***variabledecls***) ***classes*** BEGIN ***stmts*** END | SUBCOMPOSE ***id*** = id_{super} (***variabledecls***) ***classes*** BEGIN ***stmts*** END
classes	≡	| ***class***; ***classes***
expr	≡	$id_{object}.id_{mesg}$(***exprs***) | NEW (***typename***) | RELATE ***typename*** (***actuals***)
actuals	≡	| ***expr*** | ***expr***, ***actuals***

Listing 14: Syntax - Abstract Production Rules for dynamic modules.

Note that COMPOSE defines modules by an identifier (***id***), a set of variable declaration (***variabledecls***), a set of class definition (***classes***), and a set of initialization statements (***stmts***). The dynamic scope defined by COMPOSE identifies three separate contexts:

(1) The context for the entire module accessed through the reference CSELF.
(2) The context for each participant accessed within CLASS definitions of the dynamic module using SELF.
(3) The context of the actual objects accessed by invoking redefined methods after the module is instantiated using RELATE.

The following presents the extensions to the semantic domains and semantic functions relevant to dynamic modules:

Semantic Domain

val	≡ *basicval* + *loc* + *procval* + *classval* + *objval* + *composeval* + ⊥
composeval	≡ *state*→(*env* × *cenv* × *state*)

Semantic Functions

do_decl:	***decl***	→(*env* × *state*)→(*env* × *state*)
do_compose:	***compose***	→*env*→*env*
do_classes:	***classes***	→*env*→*cenv*
do_actuals:	***actuals***	→*env*→*state*→((*val* ×)* *state*)

Listing 15: Semantic Domain and Functions for dynamic modules.

The semantics of defining dynamic modules with COMPOSE and SUBCOMPOSE is similar to CLASS and SUBCLASS as shown below:

```
do_decl[compose;]⟨env, state⟩ ⇒ ⟨do_compose[compose]env, state⟩
do_compose[COMPOSE id (variabledecls) classes BEGIN stmts END]env
 ⇒ let composeval = λstate. let ⟨env1, state1⟩ =
                                    do_variabledecls[variabledecls]⟨env, state⟩
                                env2 = env;env1
                                cenv = do_classes[classes]env2
                                ⟨val, state2⟩ = do_stmts[stmts]env2 state1
                            in ⟨env1, cenv, state2⟩
   in [id→composeval]
do_compose[SUBCOMPOSE id = id_super (variabledecls) classes BEGIN stmts END]env
 ⇒  let composeval = env id_super
        composeval1 = λstate.let ⟨envs, cenvs, states⟩ = composeval state
                                 ⟨env1, state1⟩ =
                                     do_variabledecls[variabledecls]⟨env, states⟩
                                 env2 = env;envs;env1
                                 cenv = do_classes[classes]env2
                                 ⟨val, state2⟩ = do_stmts[stmts]env2 state1
                             in ⟨envs;env1, cenvs;cenv, state2⟩
     in [id→composeval1]
```

Listing 16: Semantic equations for COMPOSE and SUBCOMPOSE.

The equation for do_compose takes a COMPOSE definition and produces a new environment from ***variabledecls***, a class environment from ***classes***, and a new state from ***stmts***. The equation of SUBCOMPOSE extends the compose definition with that of the super compose identified by (***id*****$_{super}$**). In particular, SUBCOMPOSE adds new variable definitions and new class participants.

The actual binding of dynamic module participants to existing objects is done explicitly with the RELATE expression. The semantics of the RELATE expression is presented below:

```
do_expr[RELATE typename (actuals)]env state
⇒   let composeval = do_typename[typename]env
        ⟨objval_1, …,objval_n, state_1⟩ = do_actuals[actuals]env state
        ⟨env_1, cenv, state_2⟩ = composeval state_1
        [id_1→⟨clsdef_1, clspenv_1⟩,…,id_k→⟨clsdef_k, clspenv_k⟩] = cenv
        ∀i, 1≤i≤n, (⟨env'_i, state'_i⟩ = clsdef_i ((i≈1)?(state_2 | state'_i-1)))
        x = λpenv.(k≈n)?( let ∀i, 1≤i≤n,
                              ( penv_i = mixobject objval_i clspenv_i
(env_1;env'_i;[id_cself→penv])
                                  penv'_i = qualify id_i penv_i )
                          in (penv'_1;…; penv'_n) | T )
        objval = fix x
    in ⟨objval, state'_n⟩
qualify: id → penv ⇒ penv
qualify id_C penv ⇒ ∀id_i ∈ domain(penv), [id_C.id_i→penv id_i]
```

Listing 17: Semantic equations for RELATE.

RELATE binds the actual objects passed as arguments (***actuals***) to formal class participants of the dynamic module. The dynamic module (composeval) derived from ***typename*** contains a local environment (env_1), a group of classes (cenv), and the new state ($state_2$). The class environment (cenv) comprise of the formal class participants of the dynamic module. These participants ($clspenv_i$) are instantiated in mixobject (described below) and bound with the corresponding objects ($objval_i$) from ***actuals***. The result of this combination is an object whose dynamic scope comprises shared variables, local variables defined in participants, and actual objects bound to participants. Environment (env_1) denotes the shared variables, env'_i denotes the local variables of each participant, and [id_{cself}→penv] denotes the procedures defined within the module. The entire environment (env_1;env'_i;[id_{cself}→penv] is passed to mixobject. RELATE uses mixobject to produce an *objval* or $penv_i$ for each participant of the dynamic module. Each participant object is qualified using its identifier (id_i) to avoid name clashes. The final step of RELATE finds the fix point and returns the procedure environment of each participant ($penv'_i$). Note that the result of RELATE is itself an object which can participate in other dynamic modules.

The semantics of mixobject is shown below:

```
mixobject: penv → clspenv → env ⇒ pmenv
mixobject  penv clspenv env
⇒  let  x = λpenv_1.mixobjmethod penv  (clspenv (env;[id_self→penv_1]))
    in  fix x
mixobjmethod: penv → penv ⇒ penv
mixobjmethod penv penv_1
⇒  let  ∀id_i∈domain(penv_1)∩domain(penv), (procval_i = penv id_i)
    in  penv_1;[∀i(id_i→procval_i)]
```

Listing 18: Semantic equations for mixobject.

The semantic function mixobject takes the actual object (*penv*), the corresponding dynamic module participant (*clspenv*), and an environment (*env*). The arguments passed to mixobject

consists of $objval_i$ as *penv*, $clspenv_i$ as *clspenv*, and ($env_1[\boldsymbol{id}_{self} \rightarrow penv_1]$;$env'_i$) as *env*. This environment (env) is used in mixobject to bind SELF to each participant object. The fix point (fix *x*) handles any recursion. The procedures are mixed in mixobjmethod to ensure proper access to procedures in the actual object. Procedures of the actual object (penv) may override abstract procedures with identical names in the participant object ($penv_1$). Only procedures defined in $penv_1$ or overridden by penv is accessible to the participant object. Note that any "message not understood" errors resulting from the redefinitions must be detected by an appropriate type checker.

5. Conclusion

This paper has developed a denotational semantic framework for understanding the behavior of modules and classes within a programming language. This semantic framework has been used to precisely interpret two extreme behaviors of modules and their interaction with classes. On one extreme, modules are interpreted as constructs that encapsulate groups of classes. Thus, a program comprise modules, which in turn provide a local scope for class definitions. Explicit import and export clauses allow the local scope to be extended and restricted. The local scope also permits the definition of shared variables and invariants among the classes of a module. The other extreme interprets modules as dynamic entities that facilitate encapsulation of related objects. Thus, modules have a similar status as classes. In this interpretation, modules are dynamically instantiated to form objects which respond to messages. The instantiation dynamically binds actual objects to formal parts of modules. Such an approach allows the establishment of dynamic relationships between objects at run time. Dynamic modules facilitate reuse of definitions using an explicit SUBCOMPOSE construct.

There are three significant differences between these extreme interpretations. The first contrasts between a static versus a dynamic construct. A static module is used only for defining scope, and does not exist at run time. On the other hand, a dynamic module generates an object which defines a run time scope for objects. The second contrasts the granularity of the scope. The static module provides a local scope for classes, while the dynamic module provides a local scope for objects. The final difference is more subtle, and contrasts the import/export and the compose/subcompose relationships. The main difference is that import/export specifies visibility of components, while compose/subcompose copies definitions from compose into subcompose.

This work has presented the first step in understanding the behavior of classes and modules within a programming language. This semantic framework will be invaluable in answering several significant questions for languages supporting both constructs including: What is the effect of integrating a type system with classes, static modules and dynamic modules? How does subtyping relate to subclassing and subcomposing? Are other interpretations of classes and modules possible; that is, are there any interesting interpretations between the two extremes?

One major advantage of a denotational semantic framework is the ability to translate it into an equivalent executable semantics in a functional language such as ML [Wik87, Xavi92]. This translation allows validation of the semantics and testing of the intended behavior. Executable semantics for the denotational specifications presented in this paper have been developed. Several examples exercised the intended behavior. Future work is directed at developing complete semantics of the intended behavior of languages such as Modula-3 and Modular Smalltalk.

6. References

[Car88] Cardelli Luca, Donahue James, Glassman Lucille, Jordan Mick, Kalsow Bill, Nelson Greg. The Modula-3 Report. Olivetti Research California technical report ORC-1, 1988.

[Doda92] Dodani H. Mahesh and Tsai Chung-Shin. ACTS: A type system for Object-Oriented programming based on abstract and concrete classes. ECOOP '92, Lecture Notes in Computer Science, Springer-Verlag, July 1992.

[Dona89] Donahue James. Modula-3. Distinguished lecture series Volume III, University video communications, Olivetti Research California, November 29, 1989.

[Gan94] Gan Kok Siew. Denotational Semantics for compositions. Technical Reference, Department of Computer Science, University of Iowa, 1994.

[Helm90] Helm Richard, Holland Ian M and Gangopadhyay Dipayan. Contracts: Specifying behavioral compositions in Object-Oriented systems. Proceedings of ECOOP/OOPSLA '90, SIGPLAN Notices 25 (10), pages 169-180, October 1990.

[Hol92] Holland Ian M. Specifying reusable components using Contracts. Proceedings of OOPSLA, pages 287-308, 1992.

[Kam88] Kamin Samuel. Inheritance in Smalltalk-80: A Denotational Definition. Proceedings of 15th Annual ACM Symposium on Principles of Programming Languages, pages 80-87, Jan 13-15, 1988.

[Nel91] Nelson Greg. Systems Programming with Modula-3. Prentice Hall Series in Innovative Technology, 1991.

[Red88] Reddy Uday S. Object as closures: Abstract Semantics of Object Oriented Languages. Proceedings of ACM Conference on Lisp and Functional Programming, pages 1-19, July 1988.

[Ros92] Rosen J P. What Orientation should Ada Objects take? Communications of the ACM 35 (11), pages 71-76, November 1992.

[Szy92] Szyperski Clemens A. Why we need both: Modules and classes. Proceedings of OOPSLA, pages 19-32, 1992.

[Taft92] Taft S. Tucker. Ada 9X: A Technical Summary. Communications of the ACM 35 (11), pages 77-82, November 1992.

[Tsai92] Tsai Chung-Shin. ACTS: A formal model for reliable object-oriented programming based on abstract and concrete classes. Ph.D. thesis, University of Iowa, May 1992.

[Wik87] Wikstrom Ake. Functional Programming using Standard ML. Prentice Hall international series in CS, 1987.

[Wir88] Wirfs-Brock Allen and Wilkerson Brian. An overview of modular smalltalk. Proceedings of OOPSLA '88, pages 123-134, September 25-30, 1988.

[Xavi92] Xaviar Leroy and Mauny Michel. The Caml Light system, release 0.5. Documentation, September 11, 1992.