ACTS: A Type System for Object-Oriented Programming Based on Abstract and Concrete Classes

Mahesh Dodani, Chung-Shin Tsai

Department of Computer Science
The University of Iowa
Iowa City, IA 52242

Abstract. The inheritance mechanism facilitates incremental modification and is the basis for the many advantages and popularity of the object oriented paradigm. The overloading of inheritance to describe different relationship requires a semantic model to facilitate "safe" redefinition between classes. Current approaches use type systems to ensure "safe" redefinitions, and are not capable of handling all the different uses of inheritance naturally. This paper develops a uniform type system built on a hierarchy which distinguishes between abstract and concrete classes. We show how such a structure on the hierarchy can be exploited to describe all common uses of inheritance in a natural way. Finally, we develop the formalism to define abstract and concrete classes and their relationships; and the type system along with the static type checker and the semantic model to interpret type safety.

1 Introduction

The most important contribution of the OO paradigm is the inheritance mechanism. Inheritance facilitates incremental modification, and along with late binding results in the high degree of reusability promised by the OO paradigm. In the initial stages of OOP, inheritance was perceived strictly as a mechanism for code reuse; that is, allowing a class to reuse the implementation from its superclass. As the OO paradigm has matured from a programming language to a software development methodology, the inheritance mechanism has evolved from facilitating code reuse to reuse of behavior. The reuse of behavior implies the principle of substitutability [Wegner88], which states that if B inherits from A then an instance of B can always be used in any context in which an instance of A is expected.

The challenge for OO researchers is to develop the appropriate semantic model which would allow an analysis of the use of inheritance to determine whether the principle of substitutability has been maintained, thus leading to highly reliable OO system. In a seminal paper on inheritance, Wegner and Zdonik [Wegner88] identify three types of compatibility that can be obtained from inheritance.

Initial OO languages such as Smalltalk [Goldbe83] facilitated only name compatibility. Thus, objects of class B can be substituted for objects of the superclass A if class B provides the same named operations as class A. Although very flexible, this weak property can be easily violated leading to unreliable systems (that is, an object not understanding a message sent to it at run time). Although behavior compatibility is a very strong property and guarantees reliability in terms of exact conformance to the principle of substitutability, it is not feasible in practice due to two major reasons. The first is that with current technology it is very difficult (if not impossible) to prove that two semantic descriptions of behavior are compatible. The second is that behavior compatibility may be too restrictive, thereby negating the important OO property of flexibility.

Therefore research has concentrated on signature compatibility as a compromise to ensure reliable OO systems. The main objective is to impose a type system on the class hierarchy, thereby allowing an interpretation of inheritance as subtyping in the semantics of the type system [Danfor88, Cardel85]. The type checking algorithm associated with the type system is used to determine whether a given typed OO class hierarchy conforms to the semantics of the type system. If it does, then the type system guarantees type safety; that is, no "message not understood" errors at run time.

However, developing an appropriate type system for OO programming has been difficult, and the controversy surrounding this endeavor is usually depicted through the following example:

```
Point
        x: Int;
        y: Int;
        dist (): Real = (x.sqr + y.sqr).sqrt;
        eq (p:Point): Boolean
                = (x == p.getx ()).and (y == p.gety ())

ColorPoint inherits Point
        c: Color;
        eq (p: ColorPoint): Boolean
                = ((x == p.getx ()).and  (y == p.gety ())).and
                  (c.coloreq (p.getc ()))     )

p1, p2: Point
cp1: ColorPoint
p1 := new Point(10,20);
cp1 := new ColorPoint(10,20,red);
p2 := cp1;
p2.eq (p1);
```

Executing the last statement causes a run-time failure. As analyzed in [Wegner88], semantics preserving "horizontal modification" such as is done for the eq method, can *never* satisfy the principle of substitutability because "assignment of a subtype value that is not a supertype value to a supertype variable violates the principle." Note that as soon as the signature of method eq is changed from Point × Point → Boolean to ColorPoint × ColorPoint → Boolean, it is no longer possible to allow Point objects to interact with ColorPoint objects in any context that allows assignment. Point objects and ColorPoint objects can only interact in read-only mode; that is, in contexts that do not involve assignments. Thus ColorPoint objects are only read-only substitutable for Point objects. Furthermore, such a relationship between Point and ColorPoint does not facilitate a discrimination of contexts in which only properties that maintain substitutability are used; that is, contexts that use operations of Point and ColorPoint that do not include the eq operation. Note that within these contexts, ColorPoint objects are still completely compatible with (and therefore substitutable for) Point objects.

Current type systems can be divided into two broad camps according to the stand that is taken in response to the above controversy.

(A) The contravariant camp:

Type systems in this camp [Americ87, Black91, Cannin88, Cardel84, Schaff86, Mitche90, Wand89] do not allow the above controversial problem to occur by imposing a strict contravariant rule that governs modification of method argument types in the inheriting subclass. The contravariant rule enforces $T_i \leq T_i'$ and $R' \leq R$ for the redefinition of a method with signature $T_i \rightarrow R$ to $T_i' \rightarrow R'$, i=1 .. n; that is, argument types can only be generalized during redefinition. In the Point/ColorPoint example, this rule forces eq to have signature ColorPoint × Point → Boolean when redefined in class ColorPoint. Such a rule can always guarantee the principle of substitutability. Note that in this example, the burden of distinguishing between Point and ColorPoint objects

is placed on the implementation of the eq method in ColorPoint; and is usually accomplished by performing a case analysis of the type of the actual parameter.

(B) The covariant camp:

Type systems in this camp [Meyer88, Ghelli91a, Bobrow88] impose the covariant rule on method redefinition during inheritance; that is enforce $T_i' \leq T_i$ and $R' \leq R$, when redefining a method with signature $T_i \rightarrow R$ to $T_i' \rightarrow R'$, i=1 .. n in the inheriting class. Such type systems recognize that the covariant rule can lead to run-time error as depicted by the Point/ColorPoint example, and therefore provide mechanisms to handle this problem at run-time. [Meyer88] uses dynamic type checks, while [Ghelli91a] and [Bobrow88] modify the message send semantics to facilitate dynamic binding to one of the available overloaded functions. These latter type systems must provide mechanisms to map calls of eq with parameters ColorPoint × Point and Point × ColorPoint to either the function eq: Point × Point → Boolean or eq: ColorPoint × ColorPoint → Boolean.

A better perspective of the above controversy is that the inheritance mechanism is overloaded in practice in that it is used to describe the following two very different kinds of relationships between classes:

(1) Substitutable is-a relationship:

Here inheritance is being used to define a very strong relationship; that any object of class B is-a object of class A, and therefore can be substituted wherever objects of class A are expected. Establishing such a relationship necessitates enforcement of the contravariant rule for redefinition.

(2) Abstraction of common behavior:

Here inheritance is being used to abstract common behavior of a set of classes. This is a weaker relationship than the is-a, in that it relates the classes in terms of having a common interface. For instance, data structures have common behavior that allow elements to be added, removed, and enumerated. As another example, numbers have a common interface including the ability to add, subtract, divide and multiply. The basic premise here is that classes with a common abstraction either do not interact, or when they do interact then (a) the only assumption is that they can respond to the same operations, or (b) they interact within read-only contexts. Such abstractions are very common in practice, and necessitate the covariant rule for redefinition.

Thus, the controversy can be viewed from the perspective that neither the contravariant nor the covariant rule can handle this overloading of the inheritance mechanism. Any system that enforces a single rule must handle one of the above uses of inheritance as an exception. Thus, type systems of the contravariant camp handle the abstraction of behavior by forcing implementation of methods to do a case analysis, while type systems of the covariant camp handle is-a substitutability by run-time binding of possible inconsistencies during actual calls. This causes confusion when building typed OO systems, as the mechanisms provided are unnatural to one of the common usages.

It is important to acknowledge the existence of another controversy: whether the inheritance and type hierarchies should be separated or viewed as one. The proponents of the former perspective [Americ87, Black91, Cook90, Ghelli91, Graver90, Snyder87] imply that inheritance should be viewed strictly as an implementation or code reuse mechanism, and should therefore be separated from (and not confused with) the type hierarchy which is used to specify behavior compatibility. They argue that such a separation, besides clearly distinguishing between these two important kinds of incremental specification, allows a programmer the flexibility in using these mechanisms to specify precisely the relationship between objects in an OO environment. To focus the discussion on using types to approximate behavior compatibility, we shall disregard the issue of code reuse for the remainder of this paper. We feel that this kind of use of inheritance unnecessarily complicates the issue, and more importantly can easily (and more elegantly) be achieved through other mechanisms, including incorporating a Uses clause in class specifications, delegating implementations of operations to other objects, and defining and using an object of the

class whose code is to be reused as an instance variable. Note that most of the type systems that separate the inheritance and type hierarchies fall into the contravariant camp, and therefore are faced with the same problems and issues. It is however important to note that if code reuse is allowed when a type system is superimposed on inheritance, then this can lead to greater misuse of inheritance and consequently more confusing typed OO hierarchies. We concentrate on the use of inheritance as an incremental modification mechanism that facilitates reuse of behavior.

In this paper, we develop a type system known as ACTS (Abstract Concrete Type System), that recognizes the need for a hybrid approach which facilitates common uses of inheritance. We carefully analyze the practical uses of inheritance, and determine the contexts under which the contravariant and covariant rules are suitable. This analysis is done in section 2 of this paper; and leads to a categorization of classes into two distinct and disjoint sets: *abstract* and *concrete*. Such a categorization allows us to study the inheritance mechanism as it applies to relating abstract-to-abstract, abstract-to-concrete, and concrete-to-concrete classes. We prescribe the contravariant rule only on concrete-to-concrete inheritance, thereby facilitating substitutability between objects of these classes. The covariant rule is prescribed for both the abstract-to-abstract and abstract-to-concrete inheritance, implying their use as mechanisms to abstract common behavior between a set of concrete classes. Such an approach has an important side effect: we can define precise rules that guarantee well formed class hierarchies. As an example, we can define the conditions that must be met to concretize an abstract class, thus ensuring that definitions are complete. We call these rules hierarchical constraints, as they constrain the manner in which hierarchies can be formed.

The other major part of building a type system for OO programming is the development of the semantic model under which the meaning of inheritance and the property of type safety can be defined. The approaches to developing the semantic model can also be categorized according to the camp that the type system follows. Type systems that fall in the contravariant camp develop a semantic model based on traditional type theories, which rely heavily on record structures. Each class is defined by a record which defines the attributes and functions tagged with their appropriate type information. Objects are created by a constructor function, which generates object record structures where method fields are dynamically bound to the actual functions. Message passing is defined by the selection operator on object records. This semantic model lends itself very well to developing late binding and message passing. The main complications arise in the complexity of defining "self", inheritance, and subtyping. These are usually interpreted by recursive definitions, record concatenation, and complex record subtyping rules respectively. This complexity leads to limitations on inheritance and/or subtyping.

The type systems in the covariant camp are more interested in developing semantic models that facilitate flexible interpretations of message sends. In their approach objects know which class they belong to, or know their types. These type systems differentiate between mechanisms for inheritance and message passing. Inheritance is modeled as incremental modification during definition. Message passing must allow dynamic binding at run time, and is usually modelled by overloaded functions. These semantic models allow actual calls to be bound to one of the overloaded functions available due to inheritance definitions.

The semantic model for ACTS necessitates interpretation of the different kinds of inheritance possible between abstract and concrete classes. In our semantic model, abstract classes are interpreted as the set of concrete classes that belong to their inheritance hierarchy. The different kinds of inheritance are enforced initially by hierarchical constraints defined within the semantics of abstract and concrete classes, and then within the type system by transforming abstract classes into the set of concrete classes it represents. Consequently, type checking message sends is interpreted by enumerating through all possible types of the object receiving the message, and collecting all the return types of each of the function that can be applied. And since

only objects of concrete classes can exist during the execution of the program, type safety can be guaranteed by ensuring all methods in concrete classes are type-checked. Thus, our hybrid type system retains the simplicity of the covariant camp (in terms of type checking semantics) while ensuring the reliability of the contravariant camp (in terms of type safety).

The rest of this paper is organized as follows. In the next section, we motivate the need to distinguish between abstract and concrete classes, formally define abstract and concrete classes, and develop the semantic model to interpret hierarchical constraints. In section 3, we develop the type system and the underlying semantic model. We show how type checking is performed, and define the notion of type safety in our semantic model. Finally, section 5 summarizes the major contribution of this research, and outlines future plans.

2 Abstract and Concrete Classes

In this section, we develop the semantic models that define abstract and concrete classes, and allow the relationship between these classes to be established by specifying hierarchical constraints.

We start by motivating the need for separating and distinguishing between abstract and concrete classes by enumerating through possible uses of inheritance in the presence of type information. The intention here is to show the common uses of inheritance, and show how these relate to abstract and concrete classes. Note that this is in contrast to the Point/ColorPoint counter example shown in the introduction which depicts a common *misuse* of inheritance in the presence of types.

The section concludes by developing a denotational semantics for describing an OO system that incorporates abstract and concrete classes. The intention here is to provide a formal framework to precisely describe the differences and relationships between abstract and concrete classes.

2.1 Abstract and Concrete Classes in Practice

An extensive survey of the examples cited in the literature on OOP (see bibliography at the end of this paper), used in discussions on the news network (e.g., comp.lang.object and comp.lang.eiffel), and used in the development of current OO class hierarchies (e.g., Smalltalk, the NIH library, the NeXT application kit), has led to the following categorization of the common uses of inheritance in the presence of types to relate objects of two classes:

(A) Fully Compatible Behavior:

This relationship establishes a very strong property between two classes: that objects of class B can be substituted for objects of the super class A. The main intention is to facilitate interaction between objects of these classes, and to guarantee reliability between substitutions. A couple of examples will clarify the issues. Objects of class Integer are fully compatible with objects of class Fraction, as Integer objects are Fraction objects with the restriction that their denominator is always 1. However, Integer objects can provide more efficient implementations of most of the operations available (e.g., the mathematic operations including multiplication, addition and subtraction). As another example, consider the data structures Stack, Queue, and DeQueue. Addition and deletion of elements are done from one end (front or tail) in a Stack, from different ends in a Queue, and from both ends in a DeQueue. The only fully compatible relationship that exist between these data structures are that DeQueue $\leq$ Stack, and DeQueue $\leq$ Queue; that is, a DeQueue can be substituted for a Queue or a Stack in any context. (It is interesting to note that within environments that allow "code reuse", exactly the opposite relationship is obtained from an implementation perspective!) The contravariant rule is necessary to enforce this fully compatible relationship between classes.

(B) Common Protocol Behavior:

This relationship establishes a common protocol between classes; that is, the same named operations with appropriate types exist in each class. Such a relationship allows the exploitation of the uniform access to classes in promoting "interface containment" polymorphism [Cannin89a]. This kind of polymorphism is typically used to develop "generic" functions and classes that can operate over types that have a specified protocol. Typical examples of generic functions are sorts and searches, and of generic classes are data structures (such as lists and sets). The specification of generic types is usually accomplished through the use of type parameters, as is shown in the definition of a generic stack class below:

```
Parameterized Class Stack (T)
        pop: → Stack
        push: T → Stack
        top: → T
        . . .
```

The type parameter T can be further constrained by specifying the protocol that is assumed. For example, the following specification of type parameter T can constrain the use of Stacks for classes that have the protocol for points: (x: → Int, y: → Int, dist: T×T → Real, eq: T×T → T). With such a specification, both Point and ColorPoint are valid substitutions for type parameter T. Note, that Point and ColorPoint objects will never interact within such contexts, and therefore their covariant relationship (as shown at the start of the paper) is valid and safe. Several constructs and mechanisms are currently used to support common protocol behavior, including genericity [Ada83], parameterized types [Meyer88], and F-bounded polymorphism [Cannin89].

(C) Interface Compatible Behavior:

This relationship establishes a weaker property between classes: that of read-only substitutability. The only guarantee is that these classes have compatible interfaces (operations and signatures), and therefore can only interact in contexts that do not involve assignments. Note that if interaction is not allowed, then this relationship is the same as the previous common protocol behavior. Again, examples are used to clarify this relationship. Numbers (e.g., Real and Complex) have interface compatible behavior, but interactions in contexts that require assignments necessitate coercions. However, in read-only contexts, Numbers can freely interact. The same is true for Point and ColorPoint. Such interface compatible behavior is also used to reuse designs [Johnson90], which allows complete behavior definitions (that is, fully implemented) to be shared among classes. Consider, the class Collection which can completely define several operations based on the definition of operations that facilitate iteration (do:) adding (add:), and removing (remove:) as shown below:

```
Class Collection
includes: anObject
        self do: [:each | anObject = each ifTrue: [^true]].
        ^false
select: aBlock
        | newCollection |
        newCollection ← self species new.
        self do:[:each | (aBlock value: each) ifTrue:[newCollection add: each]].
        ↑newCollection
delete: . . .
do: subclassResponsibility
. . .
```

Each subclass of Collection must provide the necessary operations to inherit this behavior. Note that these classes are substitutable in contexts necessitating the Collection interface. The covariant rule is necessary to facilitate such relationships.

However, type systems that allow such a relationship must be able to discriminate contexts that are assignment free to allow interaction between objects of these classes.

As is evident from the discussion above, a type system that is based solely on a single rule (contravariant or covariant) would be incapable of handling the above uses of inheritance. One approach would be to develop separate mechanisms to handle each of the above uses, as is the case with type systems that use genericity or parameterized types or F-bounded polymorphism to handle the special case of common protocol relationship.

In ACTS, we provide a uniform solution to this problem of developing a hybrid type system capable of handling each of the above uses. We categorize classes into abstract and concrete, and then exploit the different kinds of inheritance that is possible by prescribing precise rules that guarantee one of the above relationships. Thus, a strict contravariant rule is imposed on concrete-to-concrete inheritance, thereby facilitating fully compatible relationships between classes. The relationship between abstract and concrete classes are used to tackle the other two relationships. The common protocol behavior is obtained by allowing the definition of "Selftype" in the specification of the signature of any operation in an abstract class. This "Selftype" is translated to the type of the concrete class which inherits the common protocol from the abstract class. Any other redefinition between abstract-to-abstract and abstract-to-concrete must follow the covariant rule. This rule facilitates specification of the interface compatible behavior for concrete classes. The type system semantics developed in the next section allows us to detect if such interface compatible classes are interacting within contexts that allow assignment, which are therefore tagged as potentially unsafe.

Figure 1 summarizes our typed environment, and the possible interaction between abstract and concrete classes. Note that our class hierarchies are two tiered where abstract classes are interior nodes of the hierarchy while concrete classes form the leaf nodes. An abstract class that appears as a leaf implies that it must be concretized before it can be used. Any subtree rooted by a concrete class must have all nodes as concrete classes as well.

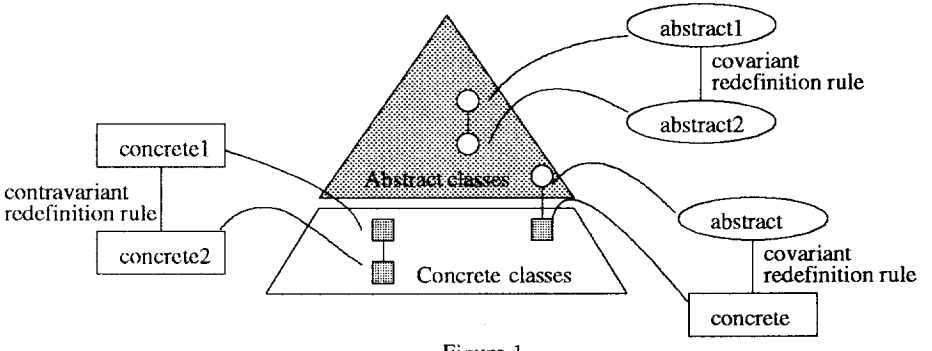

Figure 1.

2.2 Formal Specification of Abstract and Concrete Classes

The most strict requirements for an OO system supporting abstract and concrete classes can be defined by the following restrictions:

(a) An abstract class can not be instantiated.
(b) Instance variables can not be defined in abstract classes.
(c) Abstract classes must have at least one abstract method, where an abstract method is one whose definition is deferred.
(d) Non-abstract methods can not be redefined when inheriting from an abstract class.

(c) Concretization of an abstract class must provide implementations for all abstract methods.

(f) All the messages sent to the variable self must be defined.

The relaxation of any requirement is possible, and may be desirable for certain usages. For the purpose of this paper we opt for this strict definition in order to tackle the most comprehensive requirements. It is straightforward to modify the denotational semantics that is provided in this paper to accommodate any relaxations of one or more of the conditions (requirements) in this definition.

In order to develop a denotational semantics of abstract and concrete classes, we surveyed the literature to determine any available models. There exists three major denotational semantics for OOP languages developed by Cook [Cook89], Kamin [Kamin88], and Reddy [Reddy89]. We evaluated each of these semantics to determine the best for incorporating abstract classes into OOP. Cook's system is not complete for our purpose because it omits instance variables and object creation, which play important roles in our definition of abstract classes. Kamin's work is too specific in that it is done specifically for Smalltalk-80, and it covers a lot of details about message passing and block execution which are not important in our definition of abstract classes and may actually complicate our task. Reddy's work seems to cover the adequate details for our purpose and has a nice stepwise definition that can be followed easily to compensate for the inherent complication of denotational semantics. We therefore choose Reddy's definition as the basis for formalizing the notion of abstract classes.

In the following, the notation $g[x \rightarrow v]$ means a copy of the function g that maps x to v, leaving everything else unchanged. The notation $g;g'$ means updating of g with all the bindings of g'. The symbol ? is used to denote the *error* value.

Most of the definitions are straightforward. The semantics of a class is a function which defines a template (parameterized by self) for generating an object. The binding of self is accomplished only when an object is created; that is, a fixpoint is used in the definition of new e_c. The semantics of an object is a function which maps messages to methods. Note that no internal state is exposed.

The major points about the semantic definition that follows are outlined below:

- To distinguish between abstract and concrete classes (and methods) we introduce the five syntactic constructs abstractmethod, abstractclass, subabstractclass, concrete, and subtype; and the semantic domains *abstractclassval*, *abstractmethod*, *concretemethod*, *abstractclassval* and *abstractmenv*.
- Notice that the semantic domain for *abstractclassval* does not include any instance variables. Therefore, restriction (a) in the definition of abstract classes is enforceable by not allowing new to be applied to *abstractclassval*.
- The syntax allows instance variables to be defined only when abstract classes are made concrete, thereby satisfying requirement (b).
- Note that the semantic domains *abstractmenv*, *method*, and *menv* properly distinguish between abstract and concrete methods. Requirement (c) can now be enforced by checking that each abstractclass has at least one abstract method.
- Requirement (d) is checked in the definition of subabstractclass and subtype.
- The semantics of inheritance (requirement (e)) is enforced in the semantic definitions for subabstractclass and concrete. This definition enforces concretization of all abstract methods.
- Finally, notice that requirement (f) can be enforced by static binding of methods instead of dynamic binding. This is omitted for brevity.

Abstract syntax

$e ::= x$
$e ::=$ valof e
$e ::= x := e$
$e ::= e_o.m(\overrightarrow{e_a})$
$e ::=$ abstractmethod
$e ::=$ abstractclass $\{m_i(\overrightarrow{y_i}) = e_i\}$
$e ::=$ subabstractclass $e_c\{m_i(\overrightarrow{y_i})=e_i\}$
$e ::=$ concrete $e_c\ (\overrightarrow{x})\{m_i(\overrightarrow{y_i}) =e_i\}$
$e ::=$ subtype $e_c\ (\overrightarrow{x})\{m_i(\overrightarrow{y_i}) = e_i\}$
$e ::=$ new e_c
$e ::=$ hierarchy $x_1=e_1, .. x_n=e_n$ in e

Semantic domains

$\alpha \in loc$
$v, w \in val = basicval + loc + objectval +$
$\quad classval + superclassval + abstractclassval$
$\eta \in env = variable \to val$
$\sigma \in state = loc \to val$
$o \in objectval = menv$
$\varpi \in abstractmethod = state \to ?$
$\chi \in concretemethod = state \to val^* \to (val \times state)$
$\mu \in method = abstractmethod + concretemethod$
$\rho \in menv = message \to concretemethod$
$\omega \in abstractmenv = message \to method$
$\xi \in classval = state \to (menv \times state)$
$\psi \in superclassval = state \to$
$\quad (env \times (menv \to menv) \times state)$
$\zeta \in abstractclassval = state \to$
$\quad ((menv \to menv) \times state)$

Semantic mappings

$[\![$hierarchy $x_1=e_1, .., x_n=e_n$ in $e]\!]$
$=$ let $\phi = \lambda\eta.\eta_\perp[x_i \to fst([\![e_i]\!]\eta\sigma_\perp)]$
$\quad \eta = fix\ \phi$
$\quad$ in $[\![e]\!]\eta\sigma_\perp$
$[\![x]\!]\eta\sigma = \langle \eta x, \sigma \rangle$
$[\![$valof $e]\!]\eta\sigma =$ let $\langle \alpha, \sigma' \rangle = [\![e]\!]\eta\sigma$
$\quad$ in $\alpha \in loc \to \langle \sigma'\ \alpha, \sigma' \rangle$; ?
$[\![x := e]\!]\eta\sigma =$ let $\alpha = \eta x$
$\quad \langle v, \sigma_1 \rangle = [\![e]\!]\eta\sigma$
$\quad$ in $\alpha \in loc \to \langle v, \sigma_1[\alpha \to v] \rangle$;?
$[\![e_o.m(\overrightarrow{e})]\!]\eta\sigma =$ let $\langle \rho, \sigma_1' \rangle = [\![e_o]\!]\eta\sigma$
$\quad \langle \overrightarrow{v}, \sigma_2' \rangle = [\![\overrightarrow{e}]\!]\eta\sigma_1$
$\quad$ in $\rho\ m\ \sigma_2'\ \overrightarrow{v}$
$[\![$abstractclass $\{m_i(\overrightarrow{y_i}) = e_i\}]\!]\eta\sigma$
$= \langle \lambda\sigma'.$ let $\tau = \lambda\rho.\rho_\perp[m_i \to (\lambda\sigma.\lambda\overrightarrow{w}.[\![e_i]\!]$
$\quad (\eta[\overrightarrow{y_i} \to \overrightarrow{w}, \text{self} \to \rho])\sigma)]$
$\quad$ in $\tau \in menv \to ?; \langle \tau, \sigma' \rangle, \sigma \rangle$
$[\![$subabstractclass $e_c\ \{m_i(\overrightarrow{y_i}) = e_i\}]\!]\eta\sigma$
$=$ let $\langle \zeta, \sigma_1 \rangle = [\![e_c]\!]\eta\sigma$
$\quad$ in $\langle \lambda\sigma'.$ let $\langle \tau_c, \sigma_1' \rangle = \psi\sigma'$
$\quad \tau = \lambda\rho.\tau_c\rho[m_i \to$ let $\mu = \tau_c\ \rho\ m_i$
$\quad\quad$ in $\mu \in concretemethod \to ?;$
$\quad (\lambda\sigma.\lambda\overrightarrow{w}.[\![e_i]\!](\eta[\overrightarrow{y_i} \to \overrightarrow{w}, \text{self} \to \rho])\sigma)]$
$\quad$ in $\tau \in menv \to ?; \langle \tau, \sigma_2' \rangle, \sigma_1 \rangle$
$[\![$concrete $e_c\ (\overrightarrow{x})\{m_i(\overrightarrow{y_i}) = e_i\}]\!]\eta\sigma$
$=$ let $\langle \zeta, \sigma_1 \rangle = [\![e_c]\!]\eta\sigma$
$\quad$ in $\langle \lambda\sigma'.$ let $\langle \tau_c, \sigma_1' \rangle = \psi\sigma'$
$\quad \langle \eta_0, \sigma_2' \rangle = alloc\ \sigma_1'\ \overrightarrow{x}$
$\quad \eta' = \eta; \eta_0$
$\quad \tau = \lambda\rho.\tau_c\rho[m_i \to$ let $\mu = \tau_c\ \rho\ m_i$
$\quad\quad$ in $\mu \in concretemethod \to ?;$
$\quad (\lambda\sigma.\lambda\overrightarrow{w}.[\![e_i]\!](\eta'[\overrightarrow{y_i} \to \overrightarrow{w}, \text{self} \to \rho])\sigma)]$
$\quad$ in $\tau \in menv \to \langle \eta', \tau, \sigma_2' \rangle; ?, \sigma_1 \rangle$
$[\![$subtype $e_c\ (\overrightarrow{x})\{m_i(\overrightarrow{y_i} = e_i\}]\!]\eta\sigma$
$=$ let $\langle \psi, \sigma_1 \rangle = [\![e_c]\!]\eta\sigma$
$\quad$ in $\langle \lambda\sigma'.$ let $\langle \eta_c, \tau_c, \sigma_1' \rangle = \psi\sigma'$
$\quad \langle \eta_0, \sigma_2' \rangle = alloc\ \sigma_1'\ \overrightarrow{x}$
$\quad \eta' = \eta; \eta_c; \eta_0$
$\quad \tau = \lambda\rho.\tau_c\rho[m_i \to (\lambda\sigma.\lambda\overrightarrow{w}.[\![e_i]\!]$
$\quad\quad (\eta'[\overrightarrow{y_i} \to \overrightarrow{w}, \text{self} \to \rho])\sigma)]$
$\quad$ in $\langle \eta', \tau, \sigma_2' \rangle, \sigma_1 \rangle$
$[\![$new $e_c]\!]\eta\sigma$
$=$ let $\langle \psi, \sigma_1 \rangle = [\![e_c]\!]\eta\sigma$
$\quad$ in $\psi \in superclassval \to close\ \psi\ \sigma_1$; ?
$\quad close\ \psi = \lambda\sigma.$ let $\langle \eta, \tau, \sigma_1 \rangle = \psi\sigma$
$\quad\quad$ in $\langle fix\ \tau, \sigma_1 \rangle$

An executable prototype of this semantics (along with a front-end parser/scanner) has been implemented in Standard ML [Milner84]. Interested readers are referred to [Dodani92a] for details.

3 The Type System

The main intention of the type system is to provide the user (programmer) with syntactical structures and means to specify a set of classes (types), and provide a type checker that can determine the consistency of user defined types within the expressions of the program. The semantic model of the type system provides the basis to establish type safety within consistent syntactic domains. This static type consistency is established in three stages: checking consistency in the syntactical domain (e.g., ensuring hierarchical constraints), transforming entities in the syntactical domain to the semantic domain, and then ensuring consistency in the semantic model.

The following summaries the main properties of the entities involved in the type system:

(1) Types are associated with class names.
(2) Each concrete class is represented by a record structure.
(3) An abstract class is represented by the collection of concrete classes that are defined from it.
(4) An object knows its type.
(5) In the semantic model, each type corresponds to a set of objects.

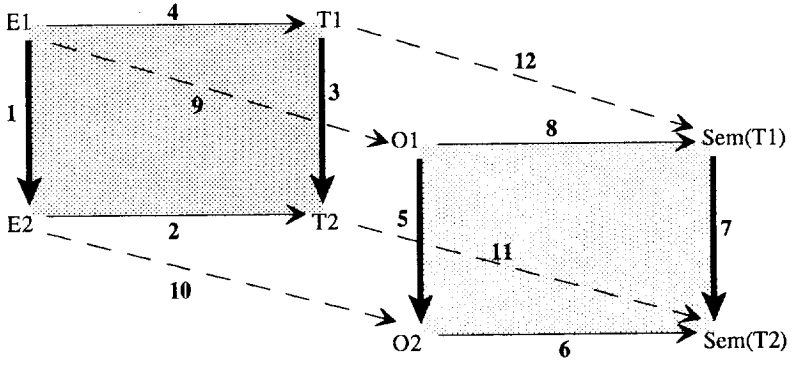

Figure 2.

The entire process of statically checking type consistency is graphically depicted in Figure 2. Vertices in this cube correspond to entities in the type system, while edges are used to represent relationship between entities (edges 1, 3, 5, 6, 7, 8), algorithms used within the type system (edges 2,4), and transformations from syntactical to semantical domains (edges 9, 10, 11, 12). Note that the syntactical domain is identified by the plane with edges {1,2,3,4}, and the semantic domain by the plane {5,6,7,8}.

We first consider the syntactical domain. The user defines expression E1 to be compatible with expression E2 (edge 1 in Figure 2) within a specified type environment. For example, the statement p1:=cp1 in the Point/ColorPoint example is interpreted as cp1 is compatible with p1. The type system builds the type environment from the user specification, and is therefore capable of inferring the type of each expression to be T1 and T2 respectively, as shown by edges 2 and 4. The syntactical subtyping rules are used to determine if T2 is compatible with T1 (edge 3) as follows:

(a) if T_1 and T_2 are both concrete classes, then $T_1 \leq T_2$,
(b) if T_1 is a concrete class and T_2 is an abstract class, then $T_1 \in T_2$, or
(c) if T_1 and T_2 are both abstract classes, then $T_1 \subseteq T_2$.

The next step is to transform the entities in the syntactic domain to entities in the semantic domain. User defined expressions are transformed using the semantic model to objects in the semantic domain (as shown by edges 9 and 10), while types are transformed into a set of objects in the semantic domain (edges 11 and 12).

The final step determines the consistency within the semantic domain. This involves ensuring that the objects belong to the set of objects represented by their type (edges 6, 8), and showing that the sets of objects represented by vertex Sem(T2) are substitutable for sets of objects of vertex Sem(T1) (edge 7). This substitutability is established by ensuring

(a) behavior compatibility between objects related by concrete classes, and
(b) set inclusion between objects related by abstract-to-abstract or abstract-to-concrete classes.

For the sake of completeness, we reproduce the entire set of definitions and theorems necessary to develop our type system in Appendix A. For the sake of brevity, we elaborate on the major components and processes involved in each of the above stages in the next subsections. In the first subsection, we describe the type expressions and the language assumed by the system, then show how to develop the class hierarchy (which includes hierarchical constraints) and type environment from a user defined specification, and then describe how to ensure consistency between the class hierarchy and type environment. The following subsection shows the main aspects of establishing consistency in the syntactic domain including the type inference rules, type checking algorithm, and the rules for establishing syntactical subtyping between types. The next subsection establishes the consistency of the process of transforming entities from the syntactic domain to the semantic domain. The final section concentrates on the semantic domain, and shows how to establish substitutability between the sets of objects defined by types. It also shows how to establish the main property of type safety.

3.1 Hierarchical Constraints and Type Environment

The type expressions are defined below. Note that AbstractClass and ConcreteClass are two mutually exclusive infinite collections of type identifiers. Subscripts are used to indicate a list, where the order is not important except in $\prod$. During type checking, expressions are assigned either Classtype or Collectiontype.

Texpr ::= Classtype
| Methodtype
| Collectiontype
| Recordtype
Classtype ::= SelfClass
| AbstractClass
| ConcreteClass
Methodtype ::= $\prod_i$ $\text{Classtype}_i \rightarrow$ Classtype
Recordtype ::= $\langle \text{Label}_i : \text{Classtype}_i , \text{Label}_j : \text{Methodtype}_j \rangle$
Collectiontype ::= $\{ \text{ConcreteClass}_i \}$

The following specifies the syntax of the novel language of ACTS that is used to define typed OO systems. The type system first builds a class hierarchy from the user specifications. The system ensures that the hierarchical constraints (that is, contravariant rule for concrete-to-concrete redefinition, and covariant rule for abstract-to-abstract and abstract-to-concrete) are maintained in the class hierarchy. This consistent class hierarchy is built according to Definitions 4 to 12 in Appendix A. The class hierarchy C_h is defined in Definition 4, while the subclassing, covariant, and contravariant relations are defined in Definitions 5, 6, and 7 respectively. Definition 8 and 9 differentiate between generalizing and specializing during modification. A type declaration system C_d which associates type expressions to classes in C_h is developed in Definition 11. Finally, the hierarchical constraints that ensures consistency between C_h and C_d is specified in Definition 12.

prog ::= hierarchy $c_1 == def_1, ..., c_n == def_n$ in *body* .
def ::= abstractclass $(m_j(y_{jk}:T_{jk}):T_j = body_j)$
| subabstractclass $T_a\ (m_j(y_{jk}:T_{jk}):T_j = body_j)$
| concrete $T_a\ (x_i:T_i)\ (m_j(y_{jk}:T_{jk}):T_j = body_j)$
| subtype $T\ (x_i:T_i)\ (m_j(y_{jk}:T_{jk}):T_j = body_j)$
body ::= *decls slist*
decls ::= decl x : T ; *decls*
| ε
slist ::= s
| s ; *slist*
s ::= e
| $x := e$ (* assignment *)
e ::= x
| b_i
| new T (* creation of object *)
| if e_b then e_t else e_f
| $e.m(\overrightarrow{e_a})$ (* message send *)

The next step is to build a type environment which facilitates static type checking from the consistent class hierarchy. This type environment is built by developing types for classes and associating types with variables. Each concrete class is represented by a record type, while each abstract class is represented by the collection of concrete classes that are defined from it. Definitions 14, 15, 16, and 17 define types for classes, show how to derive types for classes, associate types to variables, and defines the corresponding type environment respectively. The algorithm for building the type environment is shown below.

(A1) $\mathcal{A}$ [c == abstractclass $(m_j(y_{jk}:T_{jk}):T_j = body_j)$] $C_h\ C_d$
$= (\ C_h,\ C_d \cup \{\ c : \langle\ m_j:T_{jk} \rightarrow T_j\ \rangle\ \}\ )$

(A2) $\mathcal{A}$ [c == subabstractclass $T_a\ (m_j(y_{jk}:T_{jk}):T_j = body_j)$] $C_h\ C_d$
$=$ let r = $C_d(T_a)$ in $(\ C_h \cup \{\ c \leq_h T_a\ \},\ C_d \cup \{\ c : r \oplus \langle\ m_j:T_{jk} \rightarrow T_j\ \rangle\ \}\ )$

(A3) $\mathcal{A}$ [c == concrete $T_a\ (x_i:T_i)\ (m_j(y_{jk}:T_{jk}):T_j = body_j)$] $C_h\ C_d$
$=$ let r = $C_d(T_a)$ in $(\ C_h \cup \{\ c \leq_h T_a\ \},\ C_d \cup \{\ c : r \oplus \langle\ m_j:T_{jk} \rightarrow T_j\ \rangle\ \}\ )$

(A4) $\mathcal{A}$ [c == subtype $T\ (x_i:T_i)\ (m_j(y_{jk}:T_{jk}):T_j = body_j)$] $C_h\ C_d$
$=$ let r = $C_d(T)$ in $(\ C_h \cup \{\ c \leq_h T\ \},\ C_d \cup \{\ c : r \oplus \langle\ m_j:T_{jk} \rightarrow T_j\ \rangle\ \}\ )$

Note that this technique of building a type environment facilitates incremental development by allowing any predefined basic classes or class libraries to be included for consideration in type checking.

Once the type environment TEnv is built, we must show that it is consistent with the class hierarchy C_h. This relationship $x \subseteq_s t$ is established in Definition 19 and 20 as follows:

(a) x is a classtype, t is a classtype and $x \leq t$ in C_h,
(b) x is a classtype, t is a collectiontype and $x \in$ TEnv(t), or
(c) x is a collectiontype, t is a collectiontype and TEnv(x) $\subseteq$ TEnv(t).

The type environment TEnv is now used for static type checking.

3.2 Type Inference and Type Checking Algorithm

Type inference rules are given below. For simplicity we group C_h and TEnv together into a single environment Env. ($\vdash$ is the implication operator.)

(I1) For any classname or variable x:

$$\frac{\text{Env}(x) = \tau}{\text{Env} \vdash x : \tau}$$

(I2) Type subsumption:

$$\frac{\text{Env} \vdash x{:}\tau \quad \text{Env} \vdash \tau \subseteq_s \tau'}{\text{Env} \vdash x{:}\tau'}$$

(I3) Assignment:

$$\frac{\text{Env} \vdash e{:}\tau \quad \text{Env} \vdash x{:}\tau}{\text{Env} \vdash x := e{:}\tau}$$

(I4) Object creation:

$$\frac{\text{Env} \vdash T_c : \langle a_i{:}\tau_i \rangle}{\text{Env} \vdash \text{new } T_c : T_c}$$

(I5) Conditional:

$$\frac{\text{Env} \vdash e_b{:}\text{Boolean} \quad \text{Env} \vdash e_t{:}\tau \quad \text{Env} \vdash e_f{:}\tau}{\text{Env} \vdash \text{if } e_b \text{ then } e_t \text{ else } e_f{:}\tau}$$

(I6) Message send:

$$\frac{\text{Env} \vdash e{:}\ t \quad \text{Env} \vdash t{:}\ \tau \quad \text{Env} \vdash \tau.m{:}\ \vec{\sigma} \to \sigma' \quad \text{Env} \vdash \vec{e_a}{:}\ \vec{\sigma}}{\text{Env} \vdash e.m(\vec{e_a}){:}\ \sigma'}$$

where t is a concreteclass and τ is a recordtype.

$$\frac{\text{Env} \vdash e{:}\ t \quad \text{Env} \vdash t{:}\{\ t_i\ \} \quad \text{Env} \vdash t_i{:}\ \tau_i \quad \text{Env} \vdash \tau_i.m{:}\ \vec{\sigma_i} \to \sigma'_i \quad \text{Env} \vdash \vec{e_a}{:}\ \vec{\sigma_i}}{\text{Env} \vdash e.m(\vec{e_a}){:}\ \mathit{Collect}(\sigma'_i)}$$

where t is an abstractclass and τ is a collectiontype.

$$\frac{\text{Env} \vdash e{:}\ \{\ t_i\ \} \quad \text{Env} \vdash t_i{:}\ \tau_i \quad \text{Env} \vdash \tau_i.m{:}\ \vec{\sigma_i} \to \sigma'_i \quad \text{Env} \vdash \vec{e_a}{:}\ \vec{\sigma_i}}{\text{Env} \vdash e.m(\vec{e_a}){:}\ \mathit{Collect}(\sigma'_i)}$$

where *Collect* is defined as follows:
$\mathit{Collect}(\tau) = \{\ x \mid x = \tau$ if τ is a concreteclass or $x \in \text{TEnv}(\tau)$ if τ is an abstractclass$\}$

Notice that the subsumption rule facilitates substituting a subtype for a type in the rules I3 to I6 which infer types for program statements. Also notice that I4 restricts the application of new to only concrete classes, and the objects that are produced know their type. Message sends are handled in I6 by three cases depending on the type of the receiver. The first (second) rule handles the case when the receiver is some concreteclass (abstractclass). When the receiver is Collectiontype (case 3) then the message send must enumerate through every method of each type in the collection. The result is the collection of all the return types.

A type checking algorithm based on the above inference rules is developed in Definition 22 and is shown to be consistent with the inference system in Theorem 4.

3.3 Subtyping and Substitutability

Next we need to show the correspondence between subtyping in the syntactic domain and substitutability in the semantic domain. Theorem 1 below establishes the consistency between type inference and the semantics of expressions.

Theorem 1. If Env $\vdash$ E: T then the semantics of E under an environment (which is consistent with Env) is an object which is an instance of some concreteclass C such that C $\subseteq_s$ T.

Lemma 2 and Theorem 3 establish the consistency between subsumption and substitutability as follows:

Lemma 2. If C1 and C2 are two concrete classes and C1 $\leq_s$ C2 is in C_h and C1 $\leq_d$ R1 and C2 $\leq_d$ R2 are in C_d where C_d is consistent with C_h (that is, R1 specializes R2 under C_h) then any instance of C1 can substitute any value of type C2 without causing "message not understood" errors.

Theorem 3. Let C_h be a hierarchy constraint system and C_d a type declaration system where C_h is consistent with C_d. Also let Env be a type environment derived from C_d. If Env $\vdash$ $e_1 : \tau_1$, Env $\vdash e_2 : \tau_2$ and $\tau_1 \subseteq_s \tau_2$ then e_1 can be substituted for e_2 under Env without causing "message not understood" error.

3.4 Type Safety

From Theorems 3 and 4, we can define type safety (Theorem 5) as follows: If a term M is type checked (that is, assigned some type during type checking without causing the algorithm to fail) under some environment, then the execution of M under the "same" environment will not cause any "message not understood" errors.

Showing the soundness of the type system necessitates establishing the well-typeness of the program (Definition 23, 24, and 25). The well-typeness for methods and concrete classes are established as follows:

- A method specification m: $\langle a_i : \tau_i \rangle \rightarrow \tau$ = *decls slist* (in a class definition) is well-typed under a consistent set of a type hierarchy system C_h and a type environment TEnv if $\mathcal{T}$[*slist*] C_h TEnv$\oplus${$a_i : \tau_i$} $\subseteq_s$ TEnv(τ).
- A concreteclass C is well-typed under a consistent set of a type hierarchy system C_h and a type environment TEnv if every method specification that the class understands is well-typed under C_h and TEnv$\oplus${self:C, Selfclass:TEnv(C),$a_i : \tau_i$} where {$a_i : \tau_i$} is the set of all the attribute fields in TEnv(C).

From the above, we can establish that a program is well-typed if (1) the class definitions generate a consistent set of a type hierarchy system C_h and a type environment TEnv; (2) every concreteclass is well-typed under C_h and TEnv; and (3) every statement has an assigned type in C_h and TEnv$\oplus${decls}.

Thus, the type system is sound if all well-typed programs are type safe, which is established in Corollary 6 using the above definitions and Theorem 5.

4 Conclusion and Future Work

We have developed a type system for OOP that is uniform, practical, natural, and free of controversy. This type system is built by distinguishing between abstract and concrete classes, and prescribing independent rules for redefinition between each pair (abstract-to-abstract, abstract-to-concrete, and concrete-to-concrete) that guarantee type safety in the expressions of the program (that is, no "message not understood" errors at run-time). We develop a corresponding static type checker, and the necessary semantic models to ensure type safety.

An important side effect of our efforts is the development of the formal semantics to describe the ad-hoc notion of abstract classes; and more significantly the formulation of enforceable hierarchical constraints that ensure well-formed class hierarchies. We believe that such hierarchical constraints can be used as the basis for developing reliable, reusable OO frameworks.

An important issue that must be tackled by any type system for OOP is that of incremental development and compilation. Separately building the type environment from the class hierarchy allows our system to incorporate any predefined classes or class

libraries directly for type checking. However, incremental compilation can cause overheads in our system. Adding a new abstract-to-abstract or concrete-to-concrete relationship to an existing class hierarchy necessitates only type checking the classes involved in the new relationship. However, adding an abstract-to-concrete relationship requires a recompilation of all contexts in which the abstract class is used. In the worst case (for example, adding a concrete class to an abstract root class), this would require the entire hierarchy to be recompiled. Note that determining the contexts in which an abstract class is used implies the extra overhead of maintaining use chains.

We have implemented the type system in both Smalltalk [Dodani92] and Standard ML [Dodani92a]. We are continuing to analyze the applicability of the type system (for example, by redoing several class hierarchies in Smalltalk), and increasing the efficiency of our static type checker (especially as concerns incremental modification). We are also in the process of extending our type system to allow a more natural description of parameterized types, and association (like) types.

The next step in our overall effort to build a reliable OO software development environment is to facilitate formal specifications of behavior (for example, through algebraic or model-theoretic approaches); and then to build mechanisms that can enforce these behavioral constraints on implementations (for example, by converting these behavioral constraints to enforceable class invariants and pre-post conditions on methods).

References

[Ada83] U.S. Department of Defense, *Reference Manual for the Ada Programming Language*, January 1983.

[Americ87] Pierre America, "Inheritance and Subtyping in a Parallel Object-Oriented Language", ECOOP'87, pp. 234-242.

[Black91] Andrew P. Black, Norman Hutchinson, "Typechecking Polymorphism in Emerald", Digital Equipment Corporation, July 1991.

[Bobrow88] D. G. Bobrow, L. G. DeMichiel, R. P. Gabriel, S. E. Keene, G. Kiczales, D. A. Moon, *Common Lisp Object System Specification X3J13*, In SIGPLAN Notices 23 (Special Issue), September 1988.

[Cannin88] Peter Canning, Walt Hill, Walter Olthoff, "Towards a Kernel Language for Object-Oriented Programming", HP Tech. Report STL-88-21.

[Cannin89] Peter Canning, W. Cook, Walt Hill, J. Mitchell, Walter Olthoff, "F-bounded polymorphism for object-oriented programming", Proc. of Conf. on Functional Progr. Languages and Comp. Arch., 1989.

[Cannin89a] Peter Canning, William Cook, Walter Hill, Walter Olthoff, "Interfaces for Strongly-Typed Object-Oriented Programming", OOPSLA'89, pp. 457-467.

[Cardel84] Luca Cardelli, "A semantics of Multiple Inheritance", In *Semantics of Data Types* (Lecture Notes in CS, 173), 1984, pp. 51-67.

[Cardel85] Luca Cardelli, Peter Wegner, "On Understanding Types, Data Abstraction, and Polymorphism", ACM Computing Surveys 17(4), December 1985, pp. 471-522.

[Cook89] William Cook, Jens Palsberg, "A Denotational Semantics of Inheritance and its Correctness", OOPSLA'89, pp. 433-443.

[Cook90] William Cook, W.L. Hill, P.S. Canning, "Inheritance is not subtyping", 17th ACM Symposium on Principles of Programming Languages, 1990, pp. 125-135.

[Danfor88] Scott Danforth, Chris Tomlinson, "Type Theories and Object-Oriented Programming", ACM Computing Surveys 20(1) 1988, pp. 29-72.

[Dodani92] Mahesh Dodani, Chung-Shin Tsai, Tami Siu-Pui Lee, "TOPS: An Environment for Developing and Testing Type Systems for Object-

Oriented Programming Languages", submitted for consideration to OOPSLA'92.

[Dodani92a] Mahesh Dodani, Chung-Shin Tsai, "The Denotational Semantics of the Abstract/Concrete Model in SML", Technical report, Dept. of Computer Science, The University of Iowa, in preparation.

[Ghelli91] Giorgio Ghelli, "Modelling features of object-oriented languages in second order functional languages with subtypes", in Foundations of Object-Oriented Languages (G. Rozenberg ed.), Springer-Verlag, Berlin, 1991.

[Ghelli91a] Giorgio Ghelli, "A Static Type System for Message Passing", OOPSLA'91, pp. 129-145.

[Goldbe83] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[Graver90] Justin Graver, Ralph Johnson, "A Type System for Smalltalk", POPL'90, 136-150.

[Kamin88] Samuel Kamin, "Inheritance in SMALLTALK-80: A Denotational Definition", Proceedings of the 15th ACM Symposium on Principles of Programming Languages, January 1988, pp. 80-87.

[Meyer88] Bertrand Meyer, *Object-oriented Software Construction*, Prentice Hall, 1988.

[Mitche90] John Mitchell, "Toward a typed foundation for method specialization and inheritance", 17th ACM Symposium on Principles of Programming Languages, 1990, pp. 109-124.

[Milner84] R. Milner, "A Proposal for Standard ML", Proc. ACM Conf. on Lisp and Functional Programming, Austin, 1984.

[Reddy88] Uday Reddy, "Objects as Closures: Abstract Semantics of Object Oriented Languages", 1988 ACM Conference on Lisp and Functional Programming, pp. 289-297.

[Schaff86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, Carrie Wilpolt, "An Introduction to Trellis/Owl", OOPSLA'86, pp. 9-16.

[Snyder87] Alan Snyder, "Inheritance and the Development of Encapsulated Software, Components", In *Research Directions in Object-Oriented Programming*, pp. 165-188.

[Wand89] M. Wand, "Type inference for record concatenation and multiple inheritance", Porc. of LICS, 1989, pp. 92-97.

[Wegner88] Peter Wegner, Stanley Zdonik, "Inheritance as an Incremental Modification Mechanism or What Like Is and Isn't Like", ECOOP'88, pp. 55-77.

Appendix A: Definitions of the type system

Definition 1. A *record* is a finite mapping from Label to values. Label is an infinite set of symbols. Values include functions, object records, function domains, etc. which will be defined whenever needed. Notice that the fields are unordered. The notation $\langle l_i: v_i \rangle$ is used to denote a record. The selector operator (denoted by ".") can be applied to a record and returns the value of the field specified by a given label. And update operator (denoted by "$\oplus$") can be applied to two records (e.g., $\langle l_i: v_i \rangle \oplus \langle l_j: v_j \rangle$) and adds/updates every $l_j: v_j$ to $\langle l_i: v_i \rangle$.

Definition 2. *Type expressions*: see grammar in Section 3.1.

Definition 3. *Semantics of type expressions*: (each type is a set of objects)

(T1) Each concreteclass is associated with a recordtype and represents a set of objects.

(T2) Each abstractclass is associated with a collectiontype and represents a set of objects.

(T3) Each methodtype represents a function domain ($\prod_i \text{Classtype}_i \Rightarrow \text{Classtype}$).

(T4) Each recordtype represents a set of records (where each field is a function).

(T5) Each collectiontype represents a set of objects formed from $\cup$(concreteclass list).

Definition 4. An inequality of the form $s \leq_h t$, where s and t are classtypes, is said to be a *hierarchy constraint*. If for some t, $s \leq_h t$ is included in a set C of hierarchy constraints, then we say s is declared in C. A hierarchy constraint system is defined as follows:

(1) The empty set is a hierarchy constraint system.

(2) If C_h is a hierarchy constraint system and $s \leq_h t$ is a hierarchy constraint such that t is not declared in C_h, then $C_h \cup \{s \leq_h t\}$ is a hierarchy constraint system.

Definition 5. A transitive *subclass relation* $\leq_s$ in a hierarchy constraint system C_h can be defined inductively as follows:

(1) $x \leq_s t$ if $x \leq_h t$ is in C_h.

(2) $x \leq_s u$ if there exists t such that $x \leq_h t$ is in C_h and $t \leq_s u$.

Definition 6. A concreteclass x is a *concretesubclass* of t in a hierarchy constraint system C_h if there exists some s such that $x \leq_s t$ in C_h.

Definition 7. Let τ, τ', σ and σ' be type expressions. A partial order $\leq_{Covar}$ under a given hierarchy constraint system C_h on type expressions is defined as follows:

(1) For any classtype τ, $\tau \leq_{Covar} \tau$.

(2) For two classtypes σ and τ, $\sigma \leq_{Covar} \tau$ if $\sigma \leq_s \tau$ in C_h.

(3) For two methodtypes $\sigma_i \rightarrow \tau$ and $\sigma_i' \rightarrow \tau'$, $\sigma_i \rightarrow \tau \leq_{Covar} \sigma_i' \rightarrow \tau'$ if $\sigma_i \leq_{Covar} \sigma_i'$ and $\tau \leq_{Covar} \tau'$.

(4) No other $\leq_{Covar}$ relation is defined.

Definition 8. Let τ, τ', σ and σ' be type expressions. A partial order under a given hierarchy constraint system C_h on type expressions $\leq_{Contr}$ is defined as follows:

(1) For any classtype τ, $\tau \leq_{Contr} \tau$.

(2) For two classtypes σ and τ, $\sigma \leq_{Contr} \tau$ if $\sigma \leq_s \tau$ in C_h.

(3) For two methodtypes $\sigma_i \rightarrow \tau$ and $\sigma_i' \rightarrow \tau'$, $\sigma_i \rightarrow \tau \leq_{Contr} \sigma_i' \rightarrow \tau'$ if $\sigma_i' \leq_{Contr} \sigma_i$ and $\tau \leq_{Contr} \tau'$.

(4) No other $\leq_{Contr}$ relation is defined.

Definition 9. For two recordtypes $\tau \equiv \langle a_i: \tau_i \rangle$ and $\tau' \equiv \langle a'_j: \tau'_j \rangle$, if for every field $a_i: \tau_i$ in τ there is a corresponding field $a'_j: \tau'_j$ in τ' such that $a_i \equiv a'_j$ and $\tau'_j \leq_{Covar} \tau_i$ under a given hierarchy constraint system C_h, then we say record τ' *extends* τ under C_h.

Definition 10. For two recordtypes $\tau \equiv \langle a_i: \tau_i \rangle$ and $\tau' \equiv \langle a'_j: \tau'_j \rangle$, if for every field $a_i: \tau_i$ in τ there is a corresponding field $a'_j: \tau'_j$ in τ' such that $a_i \equiv a'_j$ and $\tau'_j \leq_{Contr} \tau_i$ under a given hierarchy constraint system C_h, then we say record τ' *specializes* τ under C_h.

Definition 11. An inequality of the form $t \leq_d \tau$, where t is a classtype and τ is a recordtype, is said to be a *type declaration*. If for some τ, $t \leq_d \tau$ is included in a set C of

type declarations, then we say t is declared in C. A type declaration system is defined as follows:

(1) The empty set is a type declaration system.
(2) If C_d is a type declaration system and $t \leq_d \tau$ is a type declaration such that t is not declared in C_d, then $C_d \cup \{t \leq_d \tau\}$ is a type declaration system.

Definition 12. We say a hierarchy constraint system C_h is *consistent* with a type declaration system C_d if for every hierarchy constraint $s \leq_h t$ in C_h, there exist some σ and σ' such that $s \leq_d \sigma$ and $t \leq_d \sigma'$ are in C_d and

(a) s is abstractclass and t is abstractclass, then σ extends σ' under C_h,
(b) s is concreteclass and t is abstractclass, then σ extends σ' under C_h,or
(c) s is concreteclass and t is concreteclass, then σ specializes σ' under C_h.

Definition 13. Syntax of the language: see Section 3.1.

Definition 14. A *Classtype environment* CEnv, is a finite set of the form: CEnv = { x_1: τ_1, ..., x_n: τ_n } where either

(1) x_i is a concreteclass and τ_i is a recordtype, or
(2) x_i is an abstractclass and τ_i is a collectiontype,

with no variable x_i appearing more than once in CEnv.
We write CEnv(x) = τ if x: $\tau \in$ CEnv.

Definition 15. We say a Classtype environment CEnv is *derived* from a type declaration system C_d under a hierarchy constraint system C_h if C_h is consistent with C_d and for every x declared in C_d, CEnv(x) = τ and either

(1) x is concreteclass and $x \leq_d \tau$ is in C_d, or
(2) x is abstractclass and τ is the set of all the concretesubclass of x in C_h.

Definition 16. A *syntactic type environment*, SEnv, is a finite set of the form: SEnv = { x_1: τ_1, ..., x_n: τ_n } where x_i's are variables and τ_i's are classtypes with no variable x_i appearing more than once in SEnv. We write SEnv(x) = τ if x: $\tau \in$ SEnv.

Definition 17. A *type environment*, TEnv, is the union of a Classtype environment CEnv and a syntactic type assignment SEnv. (Notice that the set of class names and the set of variables are disjoint.)

Definition 18. The algorithm that establishes C_h and C_d from class definitions: see Section 3.1.

Definition 19. We say a type environment TEnv is consistent with a hierarchy constraint system C_h if there exists a type declaration system C_d which is consistent with C_h and TEnv is derived from C_d.

Definition 20. A partial order $\subseteq_s$ on Classtype and Collectiontype under a hierarchy constraint system C_h and a type environment TEnv which is consistent with C_h is defined as follows: $x \subseteq_s t$ if either

(a) x is a classtype, t is a classtype, and $x \leq_s t$ in C_h,
(b) x is a classtype, t is a collectiontype, and x $\in$ TEnv(t), or
(c) x is a collectiontype, t is a collectiontype, and TEnv(x) $\subseteq$ TEnv(t).

Definition 21. Type inference rules: see Section 3.2.

Definition 22. The type checking algorithm is defined as follows:

In the following, C_h denotes a hierarchy constraint system and TEnv denotes a (syntactic) type environment which is consistent with C_h. The notation TEnv$\oplus${x:t} means updating the type environment with x:t. A "fail" in the algorithm terminates the algorithm abnormally (meaning that there is a type error).

(C1) $\mathcal{T}$[*decls slist*] C_h TEnv = let TEnv' = $\mathcal{TD}$ [*decl*] C_h TEnv
in $\mathcal{TSS}$ [*slist*] C_h TEnv'

(C2) $\mathcal{TD}$ [decl $x : T$; *decls*] C_h TEnv = if T is declared in C_t
then $\mathcal{TD}$ [*decls*] C_h TEnv$\oplus${ $x : T$ }
else fail

(C3) $\mathcal{TD}$ [ε] C_h TEnv = TEnv

(C4) $\mathcal{TSS}$ [s ; *slist'*] C_h TEnv = $\mathcal{TS}$ [s] C_h TEnv $\oplus$ $\mathcal{TSS}$ [*slist'*] C_h TEnv

(C5) $\mathcal{TSS}$ [s] C_h TEnv = $\mathcal{TS}$ [s] C_h TEnv

(C6) $\mathcal{TS}$ [$x := e$] C_h TEnv = if $\mathcal{TE}$ [e] C_h TEnv $\leq_s$ $\mathcal{TE}$ [x] C_h TEnv in C_h
then $\mathcal{TE}$ [e] C_h TEnv else fail

(C7) $\mathcal{TE}$ [b_i] C_h TEnv = B_i

(C8) $\mathcal{TE}$ [x] C_h TEnv = if x is declared in TEnv then TEnv(x) else fail

(C9) $\mathcal{TE}$ [new T_c] C_h TEnv = if concrete(T_c) then T_c else fail

(C10) $\mathcal{TE}$ [if e_b then e_t else e_f] C_h TEnv
= if $\mathcal{TE}$ [e_b] C_h TEnv is Boolean
then if $\mathcal{TE}$ [e_t] C_h TEnv = $\mathcal{TE}$ [e_f] C_h TEnv
then $\mathcal{TE}$ [e_t] C_h TEnv else fail
else fail

(C11) $\mathcal{TE}$ [$e.m(\vec{e_a})$] C_h TEnv
= if $\mathcal{TE}$ [e] is concreteclass
then if m is defined in Env($\mathcal{TE}$ [e_a])
then if $\mathcal{TE}$ [$\vec{e_a}$] $\leq_{sv}$ argumentsof(Env($\mathcal{TE}$ [e_a]).m)
then returnof(Env($\mathcal{TE}$ [e]).m) else fail
else fail
else collect($\mathcal{TE}$ [e],$m(\vec{e_a})$)

where $\leq_{sv}$ indicate a vector $\leq_s$, and collect(aCollectiontype,$m(\vec{e_a})$) is defined as follows:

collect({C_i},$m(\vec{e_a})$) = $\cup$ { x| x = returnof(Env(C_i).m) if m is defined in Env(C_i)
and $\mathcal{TE}$ [$\vec{e_a}$] $\leq_{sv}$ argumentsof(Env(C_i).m) }

Theorem 1. If Env $\vdash$ E: T then the semantics of E under an environment (which is consistent with Env) is an object which is an instance of some concreteclass C such that C $\subseteq_s$ T.

Proof: Induct on the syntactic constructs (of expressions).

Basis: For expressions with one constructor,

Case 1: basic values

Basic values know their types.

Case 2: variables

From (I2), if Env $\vdash$ x: τ then TEnv(x) = τ.
If x is defined in VEnv (the value environment in the denotational semantics) then VEnv(x) is either undefined (x has not been assigned a value) or it contains an instance of type τ.

Hypothesis: True for expressions with at most n constructors.

Induction: For an expression with n+1 constructors,

Case 1: assignment

Assignment does not introduce new types.

Case 2: if-then-else

No new type is introduced.

Case 3: message send

From hypothesis we know the receiving expression contains an object that understands the message. The resulting type is the collection of the return type of all possible objects that the receiving expression can contain (due to the subsumption rule). Since the type checker will guarantee that the body of the method returns an object of the specified return type, we know the resulting value is an object of the collected return types.

Lemma 2. If C1 and C2 are two concrete classes and C1 $\leq_s$ C2 is in C_h and C1 $\leq_d$ R1 and C2 $\leq_d$ R2 are in C_d where C_d is consistent with C_h (that is, R1 specializes R2 under C_h) then any instance of C1 can substitute any value of type C2 without causing "message not understood" error.

Proof: By induction on the number of message sends (during execution):
Follow from Theorem 1 and the contravariant rules.

Theorem 3. Let C_h be a hierarchy constraint system and C_d a type declaration system where C_h is consistent with C_d. Also let TEnv be a type environment derived from C_d.

If Env $\vdash e_1 : \tau_1$, Env $\vdash e_2 : \tau_2$ and $\tau_1 \subseteq_s \tau_2$ then e_1 can be substituted for e_2 under Env without causing "message not understood" errors.
Proof: (from Theorem 1 and Lemma 2)
Case 1: τ_1 is classname and τ_2 is classname
Case 1a: τ_1 is concreteclass and τ_2 is concreteclass
From definition, $\tau_1 \leq_s \tau_2$ in C_h, therefore no error from Lemma 2.
Case 1b: τ_1 is concreteclass and τ_2 is abstractclass
From definition, $\tau_1 \leq_s \tau_2$ in C_h, therefore
τ_2 is associated with a collectiontype. use Case 2 below.
Case 1c: τ_1 is abstractclass and τ_2 is abstractclass
From definition, $\tau_1 \leq_s \tau_2$ in C_h, therefore
τ_1 and τ_2 are associated with collectiontypes. use Case 3 below.
Case 2: τ_1 is classname and τ_2 is collectiontype
From definition, $\tau_1 \in \tau_2$.
Therefore, e_1 is a legitimate expression of type τ_2 .
Case 3: τ_1 is collectiontype and τ_2 is collectiontype
From definition, $\tau_1 \subseteq \tau_2$.
Therefore, there must exist some concreteclass σ such that
$\sigma \in \tau_1$ and e_1 represents an object created by σ.
(If $\sigma \in \tau_2$, use Case 2)

Theorem 4. If an expression E is assigned a type expression T during type checking then E also has inference type T under the same C_h and TEnv.
Proof: Induct on the syntactic constructs:
This follows from the fact that E and T use the same C_h and TEnv, and from the subsumption rule.

Theorem 5. If a term M is type checked (that is, assigned some type during type checking without causing the algorithm to fail) under some environment, then the execution of M under the "same" environment will not cause any "message not understood" error.
Proof: From Theorem 3 and 4, it follows that M has been assigned some type T during type checking and also has inferred type T. By inducting on the number of message sends in M it can be show that the receiver always understands the messages.

Definition 23. We say a method specification m: $\langle a_i : \tau_i \rangle \rightarrow \tau$ = *decls slist* (in a class definition) is well-typed under a consistent set of a type hierarchy system C_h and a type environment TEnv if $\mathcal{T}$[*slist*] C_h TEnv$\oplus\{a_i : \tau_i\} \subseteq_s$ TEnv(τ).

Definition 24. We say a concreteclass C is well-typed under a consistent set of a type hierarchy system C_h and a type environment TEnv if every method specification that the class understands is well-typed under C_h and TEnv$\oplus\{$self : C, SelfClass : TEnv(C), $a_i : \tau_i\}$ where $\{a_i : \tau_i\}$ is the set of all the attribute fields in TEnv(C).

Definition 25. A program is well-typed if the class definitions generate a consistent set of a type hierarchy system C_h and a type environment TEnv; every concreteclass is well-typed under C_h and TEnv; and every statement has an assigned type in C_h and TEnv$\oplus\{$decls$\}$.

Definition 26. A program is said to be type-safe if no "message not understood" will occur during the execution of the program (that is, the execution of the statements under the specified class hierarchy).

Corollary 6. The type system is sound: All well-typed programs are type-safe.
Proof: This follows from Theorem 5, and based on the fact that new only applies to user defined concrete classes, that is, only objects of concrete classes exist during the execution of the program. Note that deferred method cannot occur in concrete classes since its type is Top which is not a subtype of any user defined type, and therefore it cannot pass the type checker.