Declarative Object-Oriented Programming: Inheritance, Subtyping and Prototyping

S. Alagić, R. Sunderraman and R. Bagai

Department of Computer Science
Wichita State University
Wichita, KS 67260-0083, USA

Abstract. A design and implementation of a declarative object-oriented language is presented. The language is strongly and mostly statically typed and supports software reuse techniques such as inheritance, subtype and parametric polymorphism. It differs significantly from the existing strongly typed object-oriented languages in its declarative constraint language which is based on a suitably extended logic programming paradigm. Behavioral subtyping and advanced polymorphic facilities (such as, for example, F-bounded polymorphism) now fit naturally into this general paradigm. The underlying implementation technique produces a powerful prototyping tool for object-oriented software development and generalizes logic programming architectures by an algebraic automata based model for representing object states and state transitions.

Keywords. Object-oriented, type systems, logic programming, semantic specifications, inheritance, subtyping, polymorphism, unification, deduction.

1 Introduction

The challenges in the design and implementation of strongly and mostly statically typed object-oriented languages are in incorporating appropriate techniques for software reuse such as inheritance, subtype and parametric polymorphism, while pushing static type checking to its limits. A good illustration of the underlying issues are the controversies related to subtyping and inheritance [15, 16, 39].

Another challenge is in adopting a first order framework [9], in spite of the strong evidence that higher-order features are needed [11]. Further challenges are in designing object-oriented type systems supporting static type inference [36, 37].

Our goal is to extend the results in the development of advanced object-oriented type systems with a suitable semantic specification language which would produce a declarative, rather than a procedural, object-oriented language. In general, object-oriented languages lack high-level semantic specification facilities. Eiffel [33, 34] is probably the only fairly widely used language that has a limited assertion language. But those facilities in Eiffel do not go far enough.

Our long-term goal is to abandon procedural specification of methods altogether. That would in our opinion increase dramatically the level of software reuse. Indeed, with procedural semantics, the only way to make sure that reuse is appropriate is to look into the underlying code. In a declarative object-oriented paradigm, exploiting inheritance is thus much more convenient.

In most cases, the only practical way at the moment to specify the semantics is to write complete implementation code, and the only way to understand it is to read and/or execute it. This makes the existing strongly typed object-oriented languages particularly unsuitable for prototyping purposes. As the object-oriented paradigm is intended to be a better paradigm for managing software complexity, prototyping complex design decision involving object types (classes), inheritance hierarchies, subtype and parametric polymorphism, becomes a major design and software development issue.

In this paper we present the basic design and implementation aspects of an object-oriented, strongly typed, polymorphic language with semantic specification facilities based on logic programming. Structural properties of complex objects are represented by the features of the type system. Behavioral properties are captured by logic programming features. The language is first-order, but it accomplishes higher-order notions such as, for example, F-bounded polymorphism [11]. It is only with an appropriate semantic constraint language that behavioral subtyping [5, 27, 28] may be supported as a major generalization of what can be accomplished with object-oriented type systems, no matter how sophisticated those type systems may be.

At the moment we have only an initial implementation and the initial goal of the language is to serve as a powerful prototyping tool in designing complex object systems, even if another production quality object-oriented language (such as C++ or Eiffel) is used for the actual implementation.

The novelty in the language design is in enhancing a sophisticated object-oriented type system with logic-programming techniques for semantic specification of object types and their associated methods. The novelty in the implementation is in representing and handling state transitions in object-oriented systems using techniques of logic programming and automata theory.

The semantic specification facilities of the language are based on an appropriate form of first-order logic. Among the candidates, first-order predicate calculus appears in some proposals [35], and Horn-clause logic with equality has been used in other related work [19].

The current version of our system uses Horn clause logic with separately specified equality theory. The main advantages of this logic are that suitable unification algorithms are available [19, 24] and also that there exists a standard model (initial algebra semantics [19]) so that we know that the system is sound. The availability and the actual construction of the initial algebra semantics may also have interesting implications on the linguistic reflective features of the paradigm [20].

The main limitation of this logic is the absence of negation. Extensions of Horn-clause logic to include negation (for example, in the body) [6] and [29] have been considered. The aim of these efforts is to increase the expressibility of the language by adding some form of negation while maintaining computational tractability. At the same time, some form of denotational semantics, preferably the initial one, should still be available.

The paper is organized as follows: In Section 2, we present the basic language construct, called the specification block, as a unit of encapsulation, information hiding, inheritance, subtyping and parametric polymorphism. We also explain the major role of the logic-based specifications in our paradigm. In Section 3, we present the distinction between inheritance and subtyping in our behavioral, semantically-oriented paradigm. After a brief discussion of object creation and usage in Section 4, we discuss in detail the issues related to the conventional syntactic [12] versus the recently proposed behavioral subtyping [5, 28] in Sections 5 and 6. The contra/co-variant subtyping rules and inheritance are discussed in Section 7. The role of F-bounded polymorphism is explained in Section 8. In Section 9, we describe our prototype implementation. Its implications on information hiding are explained in Section 10. Section 11 deals with the limitations and extensions of the developed paradigm. Conclusions and comparisons with related work are presented at the end.

2 Specification Block

The language is strongly typed, polymorphic and declarative. Its main construct is the specification block as a unit of encapsulation, information hiding, inheritance, parametric and higher-order polymorphism. A specification block defines an object type (class) and includes the following components:

- Optional <u>type parameters</u>.
- A collection of <u>observers</u>. Observers are predicates, whose result type is thus omitted from the specifications.
- A collection of <u>constructors</u>. The result of a constructor application is an object with a new identity. The type of the constructed object is the same as the type specified by the specification block and is thus omitted.
- A collection of <u>mutators</u> that affect the underlying object (while preserving the object identity). The result is of the same type as that of the specification block, and is thus omitted.
- <u>Constraints</u> expressed in the chosen logic.
- <u>An equality theory</u>, preferably expressed in the equational form.

In addition, a class has functors associated with it, which when invoked, return newly created objects of that class. Functors are thus <u>creators</u>. A functor can be viewed as a method on the class itself, rather than on its objects.

We illustrate the proposed paradigm by strongly-typed, parametric, object-oriented specifications with logic-based constraints and equations. In the first example given below, the specification of the type `Natural` of natural numbers (equipped with appropriate functions such as `succ`, `add`, `sub`, `max`, and `min`) is assumed to be provided with the usual semantics.

```
Specification SimpleBag[T];
Imports Natural;
Observers
   belongs(T,Natural);
Mutators
   insert(T);
   delete(T);
Constraints
   B.insert(X).belongs(X,N.succ()) :- B.belongs(X,N);
   B.delete(X).belongs(X,N) :- B.belongs(X,N.succ());
Equality
   B.insert(X).delete(X) = B;
   B.insert(X).insert(Y) = B.insert(Y).insert(X);
End SimpleBag.
```

The above block specifies the class Bag with a type parameter T. The specification uses the type Natural, hence the Imports declaration. It defines a predicate (belongs), and two mutators (insert and delete).

Constructors are methods that return new objects while preserving the state of the underlying object, whereas mutators are methods that modify the state of the underlying object. The Constraint section contains axioms that define the effect of mutators on observers.

The language for axioms is that of Horn clause logic. As is customary in logic programs, all variables in the clauses are considered to be universally quantified over their respective types at the outside. It is impossible for any such set of axioms to be inconsistent.

Horn clauses have the following general form:

```
A :- B1, B2, ..., Bn
```

where A, B1, B2, ..., and Bn are atomic predicates. The meaning of the above clause is the following: In order for A to be true, B1 and B2 and ... and Bn must be true.

The equational theory in the above Bag specification states that deletion of an object undoes its insertion and that the order of insertions is not important. Any such user-supplied equational theory is respected by the underlying unification mechanism.

3 Inheritance and Subtyping

In this paper the term *inheritance* refers to techniques for deriving one specification from another. In some cases the derived specification will produce a subtype of the initial one. The idea behind subtyping is substitutability [12]. We say that $T_2 <: T_1$ (T_2 is a subtype of T_1) if an instance of T_2 may be substituted any place an instance of T_1 is expected. If S_1 and S_2 are specifications, then S_2 will define a subtype of the type defined by S_1 iff for every method $M(A_1, A_2, \ldots, A_n) : A$ in S_1, S_2 has a method $M(B_1, B_2, \ldots, B_n) : B$ such that $A_i <: B_i$ for all i

(contravariance) and $B <: A$ (covariance). The subtyping relation is reflexive and transitive.

This is the usual definition of subtyping [12] which we call in this paper syntactic. A type system can only enforce syntactic subtyping in the sense that type errors will be detected if such substitutions are permitted. Although very important (and in general impossible to check at compile time in most non-trivial strongly typed object-oriented languages), syntactic subtyping addresses only a small portion of the substitutability issue. Since the object-oriented paradigm is a behavioral paradigm, a stronger and perfectly natural requirement is that if a substitution is performed, a user viewing an object of type T_2 as an object of type T_1 should see no behavioral difference [28]. Such a requirement can be addressed only in a paradigm that extends a sophisticated object-oriented type system with semantic (behavioral) specification facilities.

As an example, consider the specification BoundedBag derived by inheritance from SimpleBag.

```
Specification BoundedBag[T];
Inherits SimpleBag[T], Redefines belongs;
Observers
   non_full();
   size(Natural);
Mutators
   clear();
Constraints
   B.clear().belongs(X,0);
   B.clear().size(0);
   B.clear().non_full();
   B.insert(X).belongs(X,N.succ()) :- B.non_full(), B.belongs(X,N);
   B.delete(X).non_full() :- B.belongs(X,N.Succ());
   B.insert(X).size(X,N.Succ()) :- B.non_full(), B.size(X,N);
   B.delete(X).size(N) :- B.size(N.succ()), B.belongs(X,N.succ());
End BoundedBag.
```

In addition to the inherited predicates, mutators and constraints, BoundedBag introduces new predicates non_full and size (reflecting its bounded nature), and an additional mutator clear, which empties the bag[1].

The effect of the new mutator on both inherited (belongs) and new observers (non_full and size) must be defined. In addition to that the effect of the inherited mutators (insert and delete) on the new observers (non_full and size) must be provided.

Syntactically, BoundedBag is a subtype of SimpleBag, i.e. BoundedBag[T] <:

[1] An observant reader will notice that the predicate non_full is not completely specified. The specification could be completed if internal implementation related aspects are included such as a private predicate bound with the associated constraints. We do not elaborate this further in this paper, but the issue is related to internal interfaces as proposed in [35] or 'friend' functions in C++.

SimpleBag[T] for any specific T [2]. Semantically or behaviorally, this is not the case, as the effects of the mutators of SimpleBag have been redefined. As a result, if a bounded bag is viewed as a simple bag, it may exhibit behavior unexplicable from the simple bag specification.

Our approach is not based on Hoare-style pre and post conditions as those are, in our opinion, suitable for procedural paradigms. In spite of that, it is possible to make some comparisons with contravariant/covariant rules as they apply to pre and post conditions [28]. Consider the constraints expressing the effects of the insert mutator on the belongs observer in the Bag and BoundedBag specifications.

```
B.insert(X).belongs(X,N.succ())  :- B.belongs(X,N);
B.insert(X).belongs(X,N.succ())  :- B.non_full(), B.belongs(X,N)
```

We have B.non_full, B.belongs(X,N) implies B.belongs(X,N), which is exactly the opposite of what the contravariant/covariant rule requires as explained in [28]. Hence, BoundedBag is not a behavioral subtype of SimpleBag.

Consider now another specification derived from SimpleBag by inheritance. In the specification of Bag abstraction given below, a collection of new constructors is introduced: union, intersect, difference. Additional constraints are provided defining the effects of the constructors on the observers (only one in this case, belongs). These constraints define what can be observed about the state of a newly created object.

```
Specification Bag[T];
Inherits SimpleBag[T];
Constructors
   union(Bag[T]);
   intersect(Bag[T]);
   difference(Bag[T]);
Constraints
   B1.union(B2).belongs(X,M.max(N))  :- B1.belongs(X,M),
                                        B2.belongs(X,N);
   B1.intersect(B2).belongs(X,M.min(N))  :- B1.belongs(X,M),
                                            B2.belongs(X,N);
   B1.difference(B2).belongs(X,M.sub(N))  :- B1.belongs(X,M),
                                             B2.belongs(X,N);
End Bag.
```

Bag is both a syntactic and a semantic (behavioral) subtype of SimpleBag. Just adding new methods in general produces a syntactic subtype. In order to get a behavioral subtype, the effects of new mutators must be explained in terms of the existing ones [28]. But no mutators were added, just pure constructors. If union, intersection, difference were defined as mutators, additional constraints would be required that (equationally) define those mutators in terms of the existing ones (insert, delete). Only if such a definition is possible, would we have a behavioral subtype [28]. This problem is elaborated in Section 6.

[2] But observe that T2 <: T1 does not imply BoundedBag[T2] <: SimpleBag[T1] or else problems discussed in [15] would occur.

As a consequence, a bag can be used any place a simple bag is expected. The type system will detect no problem nor will the behavior of such a bag viewed as a simple bag produce unexpected effects.

4 Object Creation and Usage

Object creation is handled by separate constructs called *functors*, as in [35]. But a major difference in comparison with [35] is that the initial state of an object in our approach satisfies the constraints present in the functor definition. Observe the underlying nondeterminism. The constraints in general do not determine a specific state of an object, but rather a collection of states satisfying the constraints. For example, only the values of some instance variables may be specified or constrained. Because of all this, we allow more than one functors for any specification block. Different functors associated with the same specification block would in general create objects with different initial states.

The following is a specification of a functor that returns a bag of elements of type T. A bag returned by this functor is always empty [3].

```
Functor null[T]: Bag[T];
Constraints
   null[T].belongs(X,0);
End null.
```

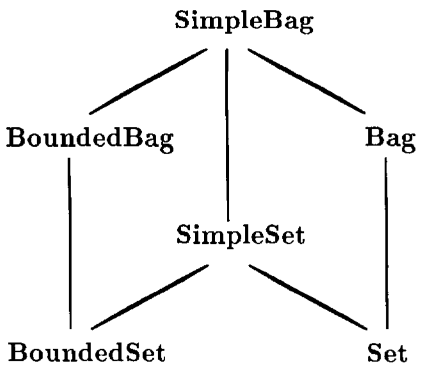
An example usage of the above specifications might look like this:

```
let Groceries1 = null[GroceryType];
let Groceries2 = null[GroceryType];
...
let CommonGroceries = Groceries1.intersection(Groceries2);
let AllGroceries = Groceries1.union(Groceries2);
```

5 Behavioral Subtyping and Multiple Inheritance

The following diagram illustrates various possibilities for deriving one specification from another as they apply to bags and sets.

[3] Scope rules are defined in such a way that all the observers of the corresponding specification are imported into the scope of a functor associated with that specification. This allows specification of the observable properties of the newly created object to be defined in the functor



If we allow selective inheritance, other options would be possible, such as deriving SimpleSet from BoundedBag and SimpleSet from Bag. We feel that the above diagram presents semantically reasonable extensions. It also includes interesting examples of multiple inheritance. Indeed, BoundedSet could be defined by multiple inheritance from BoundedBag and SimpleSet. Likewise, Set may be defined by multiple inheritance from SimpleSet and Bag. As our paradigm is behavioral (in addition to being syntactic), an interesting issue is how the associated logic-based paradigm handles behavioral aspects of multiple inheritance demonstrated by the above examples. This important topic is discussed in Section 11.

SimpleSet is derived by inheritance from SimpleBag. Sets differ from bags in that they may contain any element only once. The effects of the inherited mutators on the new observer (element) must be defined. However, the effect of the inherited mutators on the inherited observers is redefined as well, as sets do not behave as bags when those mutators are applied. Inserting an element into a set more than once has no effect, quite contrary to the behavior exhibited by a bag. Deleting an element also behaves differently.

A shortcut is used in the SimpleSet specification block. The new observer element is defined in terms of the inherited one (belongs) and the inherited one is redefined.

```
Specification SimpleSet[T];
Inherits SimpleBag[T] Redefines belongs;
Observers
   element(T);
Constraints
   S.element(X) :- S.belongs(X,1);
   S.insert(X).belongs(X,1);
   S.delete(X).belongs(X, 0) :- S.belongs(X,1)
End SimpleSet.
```

The result is a syntactic, but not a semantic subtype. Substituting a simple set object any place a simple bag object is expected will cause no problem as far as the type system is concerned. However, the behavior of such a simple bag will not be bag-like. In fact, it will be set-like.

A specification of the abstraction BoundedSet may now be derived in two possible ways. One way is to derive it from BoundedBag. BoundedSet becomes a syntactic subtype of BoundedBag. However, it obviously is not a semantic subtype.

Another way to derive a specification for BoundedSet is from SimpleSet, adding new predicates non_full and size and a new mutator clear.

```
Specification BoundedSet[T];
Inherits SimpleSet[T];
Observers
   size(Natural);
   non-full();
Mutators
   clear();
Constraints
   S.clear().belongs(X,0);
   S.clear().size(0);
   S.clear().non-full();
   S.insert(X).belongs(X,1)  :- S.non-full();
   S.delete(X).non_full()  :- S.Belongs(X,1);
   S.insert(X).size(X,N.succ())  :- S.non_full(), S.size(X,N),
                                    S.belongs(X,0);
   S.delete(X).size(N)  :- S.size(N.succ()), S.belongs(X,1)
End BoundedSet.
```

The above specification produces a syntactic, but not a behavioral subtype.

6 Behavioral Subtyping and Equational Constraints

For an example of behavioral subtyping consider the following stack specification. The class Stack will form a supertype of the class List specified immediately after.

```
Specification Stack[T];
Observers
   top(T);
Mutators
   push(T);
   pop();
Constraints
   S.push(X).top(X);
Equality
   S.push(X).pop() = S;
End Stack.
```

The mutators push and pop affect the state of the underlying stack, and the predicate top(X) is true if X is the topmost element of the underlying stack object. top is defined only for mutator invocation sequences involving push.

Let us now consider the following list specification:

```
Specification List[T];
Inherits Stack[T]; Imports Natural;
Observers
   car(T) renames top;
   is_nil();
   length(Natural);
Mutators
   cons(T) renames push;
   cdr() renames pop;
   concat(List[T]);
Constraints
   L.is_nil() :- L.length(0);
   L.cons(X).length(N.succ()) :- L.length(N);
Equality
   L1.concat(L2) = L2    :-   L1.is_nil();
   L1.cons(X).concat(L2) = L1.concat(L2).cons(X);
End List.
```

The class List inherits all methods, constraints and equality axioms from the class Stack. However, the methods push, pop and top are renamed to cons, cdr and car, respectively. Although this presents a minor problem for the type system, it is a major convenience for the users and in fact required in cases of multiple inheritance. An important point to observe is that the new mutator concat introduced in List specification may be entirely equationally expressed in terms of the existing ones. Since the predicates is_nil and length are also specific to lists, the specification contains clauses for them as well.

7 Contravariant Subtyping and Covariant Inheritance

A specification for Set may now be derived in two ways: from SimpleSet or from Bag. There is nothing unexpected in the derivation of Set from SimpleSet. Set becomes both a syntactic and a semantic subtype of SimpleSet. Deriving Set from Bag produces a well-known controversy [15], except that our semantic framework makes the discussion much more general.

```
Specification Set[T];
Inherits Bag[T] Redefines belongs;
Observers
   element(T);
Constraints
   S.element(X) :- S.belongs(X,1);
   S.insert(X).belongs(X,1);
   S.delete(X).belongs(X,0) :- S.belongs(X,1);
   S1.union(S2).belongs(X,1) :-  S1.element(X);
   S1.union(S2).belongs(X,1) :- S2.belongs(X,N);
   S1.intersect(S2).belongs(X,1) :- S1.element(X), S2.belongs(X,N);
   S1.difference(S2).belongs(X,1) :- S1.element(X), S2.belongs(X,0);
End Set.
```

The constructors now take arguments of type `Bag[T]`. The type of the underlying object, as well as the type of the constructed object is `Set[T]`. The contravariant rule for the argument types of methods is satisfied and so is the covariant rule for the result type, as explained in Section 3. For example, the signature for union is now `union(Bag[T]): Set[T]` and `Set[T]` <: `Bag[T]`. Set defined in this way indeed becomes a syntactic subtype of Bag. It is obviously not a semantic subtype, because the rules for the mutators have been redefined. Although the type system would have no problem with the above definition of Set, a user viewing a set as a bag will see a different, non bag-like behavior.

Quite contrary to the usual contravariance of the argument types [12, 15], consider a covariant redefinition of the signatures for the Bag constructors, given below:

```
Specification Set[T];
Inherits Bag[T] Redefines belongs,union,intersect,difference;
Observers
   element(T);
Constructors
   union(Set[T]);
   intersect(Set[T]);
   difference(Set[T]);
Constraints
   S.element(X) :- S.belongs(X,1);
   S.insert(X).belongs(X,1);
   S.delete(X).belongs(X,0);
   S1.union(S2).belongs(X,1) :- S1.element(X);
   S1.union(S2).belongs(X,1) :- S2.element(X);
   S1.intersect(S2).belongs(X,1) :- S1.element(X), S2.element(X);
   S1.difference(S2).belongs(X,1) :- S1.element(X), S2.belongs(X,0)
End Set.
```

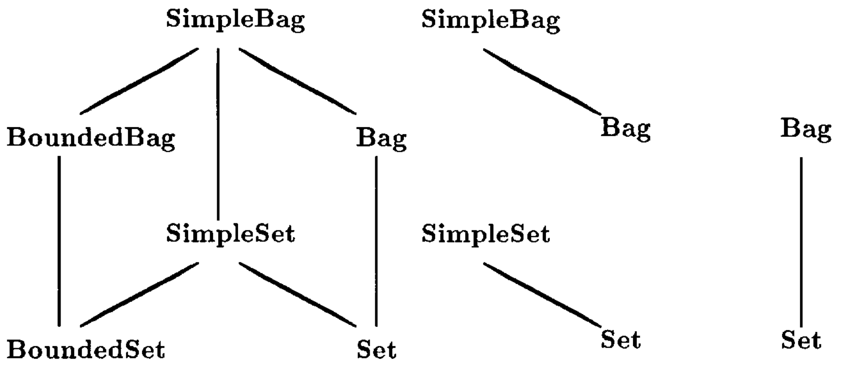
The above redefinition is perfectly intuitive and in fact in the style of Eiffel [34]. But now the signatures for the constructors do not satisfy the contravariant requirements for the argument types. Both the argument and the result types have been extended covariantly. It is now possible to write programs that type check at compile time and fail at run time due to a type error. The following is an example:

```
S1,S2: Set[Natural];
B1,B2: Bag[Natural];
...
Let  B1=S1;
Let  S2=B1.union(B2);
```

The problem comes from the local nature of type checking and dynamic binding. `B1.union(B2)` obviously type checks at compile time. But at run time, the redefined version of union will be chosen, since the method is applied to an instance of type `Set[Natural]` (dynamic binding). The selected method will get an argument of the supertype, contrary to the principle of substitutability, according to which only an instance of a subtype may be substituted in place of its supertype.

Set as specified above is neither a syntactic nor a behavioral subtype of Bag. Not only have the signatures of the constructors union, intersect and difference been redefined in such a way that the contravariant subtyping rules for argument types are violated, but at the same time there are behavioral changes. The effects of the inherited mutators insert and delete have been redefined. This is thus an example of pure inheritance where no subtyping, either syntactic or semantic occurs. In spite of that, the derived abstraction makes perfect sense. But no substitutability, either on syntactic or on semantic grounds, is appropriate.

We summarize our analysis in graphical form:



Syntactic Subtyping *Semantic Subtyping* *Covariant Inheritance*

8 Higher-order Polymorphism

The main language construct, the specification block, comes in general with a type parameter. That allows support of higher-order features in spite of the fact that the language is still first order. A parametric specification block is in fact a function TYPE → TYPE where TYPE is the collection of all types each of which is expressed by a specification block. TYPE, however, is not explicitly present in the language. It is constructed starting with specification blocks for types such as Natural, Real, String etc. which are in fact predefined and thus have fixed interpretations.

An important form of higher-order polymorphism for object-oriented languages is F-bounded polymorphism [11]. Although a fairly abstract and higher-level notion, it actually performs a very pragmatic role in our language. Here is an illustration. To specify an ordering we need at least a predicate, let us call it less_than. Its generic (parametric) definition is given below.

```
Specification Order[T];
Observers
   less_than(T);
End Order.
```

A specific ordering will now be given by introducing appropriate constraints. So, for example, a preorder is just reflexive and transitive, as specified below. But the type parameter of the preorder abstraction must satisfy the condition T <: Order[T]. This condition guarantees that whatever the underlying type T is, it must have the predicate less_than. The form of subtyping is not just bounded by a particular specification. Rather, an F-bound is Order[T].

```
Specification PreOrder[T <: Order[T]];
Constraints
   X.less_than(X);
   X.less_than(Z) :- X.less_than(Y), Y.less_than(Z);
End PreOrder.
```

Although F-bounded polymorphism [11] is important for object-oriented type systems as it allows some inheritance relationships to be captured within the type system, it is only within the framework of a constraint language that it assumes its full meaning. In the above example, it is only when the F-bounded subtyping condition is satisfied that the universal quantification of all variables (standard in logic programming and adopted in our language design) makes sense.

Following the same logic we can define the class of ordered sets as follows:

```
Specification OrderedSet[T <: PreOrder[T]];
Inherits Set[T];
...
End OrderedSet.
```

The above definition guarantees that the type parameter T is equipped with the predicate less_than that satisfies the axioms for preorder. Recall that our definition of subtyping <: given in Section 6 is a semantic rather than the usual [12] syntactic one.

9 Prototype Implementation

The goal of the current implementation is not a production quality support for a declarative object-oriented programming language. Rather, the goal is a powerful prototyping tool based on an advanced type system extended with logic-based semantic specification facilities. Because of that we explore suitable logic-programming architectures and generalize them so that they become appropriate for typed object-oriented environments.

Two major components of the underlying support for any logic programming system are the unification algorithm and an implementation of some kind of resolution. Augmenting clauses with an equational theory complicates matters as ordinary unification can no longer be used. In [38], it was shown that one can work on the clauses alone and yet have a complete inference system in a theorem-prover, if a generalized unification algorithm is used, one which respects the equational theory in question.

One such unification algorithm is presented in [24] where the process of unification is seen as solving equations within the equational theory. Another practical approach to logic programs with equality theories is presented in [26] where an extension to Prolog called Prolog-with-Equality is presented. Here, when two terms do not unify syntactically, an equality constraint is used to attempt to prove the two terms equal. We use similar techniques in the implementation of the deductive capabilities of our language.

The first step in implementing our object-oriented language on an underlying logic programming architecture is to translate features from the former paradigm into the latter. We first present a simple translation procedure and then describe the run-time organization of the deductive system.

9.1 Translation

Associated with each object class in a given program is a collection of its observer, constructor and mutator methods. For any n-ary predicate method p belonging to a class c, we introduce an $(n+1)$-ary predicate symbol p_c. The arity of p_c is one more because the underlying object of p is now an argument supplied to p_c. We attach the class name with the predicate symbols to distinguish between identical predicate methods in different classes. This reflects the many sorted (in fact order-sorted) nature of the paradigm [19]. Similarly, for any n-ary mutator method f belonging to a class c, we introduce an $(n+1)$-ary function symbol f_c.

The constraints in any class are also translated into standard Horn logic syntax in a straightforward way. For example, the constraint

```
B.insert(X).belongs(X,N.succ()) :- B.belongs(X,N);
```

in the class Bag now becomes

```
belongs_Bag(insert_Bag(B,X),X,succ_Natural(N)) :-
                                          belongs_Bag(B,X,N).
```

The equality axioms are translated similarly into unit Horn clauses (ones having empty bodies). A functor r that returns an object of class c now becomes a constant r_c. Its constraints are also translated as above.

9.2 Objects and States

At run-time user(s) perform actions that may create, delete or manipulate objects. The object states are identified with equivalence classes of ground terms representing compositions of messages (mutator invocations) each one beginning with a functor application. This model is in fact based on a technique from algebraic automata theory.

For example, the action `let x = null[Natural]` creates a bag of natural numbers, whose current state is the term `null_Bag`, and since it satisfies the axioms in the `null` functor specification, the term represents the empty bag of natural numbers.

The action `x.insert(0)` inserts the element `0` in `x` by modifying its state to the term `insert_Bag(null_Bag, 0_Nat)`. And if `y` is also a bag of natural numbers with the current state

```
insert_Bag(insert_Bag(null_Bag, 0_Natural), 1_Natural)
```

then the action `let z = x.union(y)` makes the current state of the object `z` to be the term

```
union_Bag(insert_Bag(null_Bag, 0_Natural),
          insert_Bag(insert_Bag(null_Bag, 0_Natural),
                     1_Natural))
```

(The constant `1` is considered to be equated to the term `0.succ()` in the specification of `Natural`.)

Queries can be made by invoking predicate methods. For example, the query

```
z.belongs(0, N)}
```

produces the solution `N = 0.succ().succ()` indicating that the element `0` belongs 2 times to the object `z`.

9.3 Run-time Deduction

As suggested in the above example, constraints in a class specification are used only for query answering, much like in standard logic programs. However, the unifications performed in the derivations respect the equality theories contained in the specifications.

For an example of the modified unification, consider the equality axioms contained in the specification of the class `Bag`:

```
B.insert(X).delete(X)  = B;
B.insert(X).insert(Y)  = B.insert(Y).insert(X)
```

Let b_1 and b_2 be bags with the following histories of method invocations:

$$b_1 = \texttt{null.insert(0).insert(1)}$$
$$b_2 = \texttt{null.insert(1).insert(2).insert(0).delete(2)}$$

In other words, b_1 is a bag, which was initially empty and into which the elements `0` and `1` were inserted, in that order. And b_2 is a bag, which was also initially empty, and into which the elements `1`, `2` and `0` were inserted in that order, and the element `2` was later deleted. These two terms in fact belong to the same equivalence class and thus represent the same state. The equivalence relation is defined somewhat differently than in [19]. We say that terms t_1 and t_2 are equivalent iff $t_1 = t_2$ is provable from the given set of equations E .

Treating the equality axioms as rewrite rules, we can have the following rewritings of the terms b_1 and b_2:

b_1 = `null.insert(0).insert(1)`
$\rightarrow$ `null.insert(1).insert(0)`

b_2 = `null.insert(1).insert(2).insert(0).delete(2)`
$\rightarrow$ `null.insert(1).insert(0).insert(2).delete(2)`
$\rightarrow$ `null.insert(1).insert(0)`

We thus see that b_1 and b_2 unify with respect to the equality axioms and thus represent the same state of the underlying object.

If the equality axioms in the specified classes, when considered as rewrite rules, are confluent and terminating, then the following unification algorithm proposed in [18] and improved in [22] is known to be complete, i.e. it produces a complete set of unifiers.

Algorithm E-UNIFY: Let t_1 and t_2 be terms and E be a confluent and terminating set of rewrite rules. Let τ be a new function symbol and let $s_0 = \tau(t_1, t_2), s_1, \ldots, s_n = \tau(t_3, t_4)$ be a sequence of terms such that:

- for each i, $0 \leq i < n$, s_i contains a non-variable subterm w_i that unifies (in the ordinary sense) with the left hand side of a rule $l_i = r_i$ in E with most general unifier θ_i, and s_{i+1} is obtained from s_i by replacing w_i by $\theta_i(r_i)$; and
- t_3 and t_4 unify (in the ordinary sense) by a substitution α.

Then generate the substitution $\theta_0\theta_1 \cdots \theta_{n-1}\alpha$ as an E-unifier of t_1 and t_2. □

The above algorithm is known to generate a complete set of E-unifiers for t_1 and t_2 (see [18, 22, 19]), i.e. each substitution generated by it is an E-unifier and, for any E-unifier γ of t_1 and t_2, there is some E-unifier θ generated such that for some E-unifier δ, $\gamma = \theta\delta$.

A *derivation* can now be defined as a (finite or infinite) sequence $G_0, G_1, \ldots$ of goals such that, for each i,

- G_i is a set of the form $\{B_1^i, \ldots, B_{m_i}^i\}$, where each B_j^i is an atom;
- there is a constraint clause of the form

$$A^i :- D_1^i, \ldots, D_{n_i}^i$$

  with variables renamed to names never before used in the derivation;
- θ_i is an E-unifier of A^i and B_j^i, for some j, $1 \leq j \leq m_i$; and
- G_{i+1} is the set

$$\{B_1^i, \ldots, B_{j-1}^i, D_1^i, \ldots, D_{n_i}^i, B_{j+1}^i, \ldots, B_{m_i}^i\}\theta_i.$$

A derivation is *successful* if some goal in it is empty. The composition of the unifiers in that derivation up to the empty goal produces the desired solution in the usual way. Observe that unlike standard logic programs, here it is possible to have more than one E-unifiers at each step. (In fact, for some pairs of terms there are an infinite number of E-unifiers.) However, the set of all solutions is still recursively enumerable, and can thus be effectively enumerated.

10 Information Hiding

Information hiding is supported in our paradigm in a way that the actual object state is never exposed. The only way for a user to observe properties of the hidden object state is by invoking the observer methods, and the only way to change that state is by invoking mutators. In fact, two states of an object may in fact be different in the underlying implementation, and still produce the same observable behavior. Those two states will then be undistinguishable by the users and thus belong to the same equivalence class of states in the underlying implementation.

Given the particular example, we can illustrate such situations. For Bag and Set specifications, an immediate question is whether the familiar axioms such as idempotence, commutativity and associativity, should be given as additional equational constraints, as in [19].

If B1, B2 and B3 are bags of type Bag[T], the standard equational axioms for union are:

```
B1.union(null[T])=B1
B1.union(B1)=B1
B1.union(B2)= B2.union(B1)
B1.union(B2.union(B3))= B1.union(B2).union(B3)
```

The above properties of the constructors associated with the bag abstraction are not given in the specification Bag[T]. Moreover, at the level of logic terms, the above properties do not hold. Since this is not observable via the predicate methods in the class Bag, explicit mention of these properties is not necessary.

As an example, let b1 and b2 be singleton bags of natural numbers with elements 1 and 2, respectively. Consider

```
let b12 = b1.union(b2);
```

Then, the state of the bag b12 after the above action is:

```
union_Bag(insert_Bag(null_Bag, 1), insert_Bag(null_Bag, 2))
```

Likewise, the desired action and the corresponding state of the bag b21 are:

```
let b21 = b2.union(b1);

union_Bag(insert_Bag(null_Bag, 2), insert_Bag(null_Bag, 1)).
```

The states of the bags b12 and b21 are not equal with respect to the equality theory. However, the two bags behave in exactly the same way with respect to all the observers contained in the Bag specification.

11 Limitations and Extensions of the Paradigm

Although the paradigm in which the constraints are expressed in Horn clause logic with separate equational axioms is intuitively attractive and computationally tractable, there are situations where we need additional features such as (1) equality in the clauses, (2) negation in the body of the clauses, and possibly (3) disjunctions in the head of the clauses. We shall illustrate these situations with examples and suggest how we are extending our paradigm to handle them.

11.1 Equality in Clauses and Axioms

The following example specification of PartialOrder has equality in the head of a constraint clause:

```
Specification PartialOrder[T <: Order[T]];
Constraints
   X.less_than(Z) :- X.less_than(Y), Y.less_than(Z);
   X = Z :- X.less_than(Y), Y.less_than(X);
End PartialOrder.
```

This is a simple and easily tractable extension of our paradigm where all the implementation techniques of the previous section remain applicable. For details, see also [19].

11.2 Stratified Negation

A set of clauses is said to be *stratified* if the clauses can be partitioned into ordered sets of clauses such that if a negated atom appears in the body of a clause in a partition, then the definition of that atom appears in a previous partition and if a positive atom appears in the body of a clause in a partition, then its definition either appears in the same partition or a previous partition. An example specification that requires stratified negation follows.

Consider the specification of the proper_subset predicate of the Set abstraction. The signature for proper_subset predicate would have the form:

```
proper_subset(Set[T])
```

and the associated constraint is:

```
S1.proper_subset(S2) :- S1.subset(S2), ~S1.difference(S2).empty()
```

A collection of stratified clauses does have a minimal model semantics as described in [7]. The intended minimal model for a collection of stratified clauses is constructed in an intuitive manner as follows: starting from the first partition of clauses, compute the logical consequences of the clauses in a partition using only the consequences obtained in previous partitions. Since the clauses are stratified, the predicate that appears in a negated atom will have been completely computed (i.e. both its positive and negative ground instances would be known) in a previous partition and hence can be used in the current partition. The derivation of negative facts for a partition is done by complementation after all the positive facts have been computed for the partition.

11.3 Non-stratified Negation

There are situations where stratified negation is not sufficient. We now present an example.

The `difference` and `symmetric_difference` constructors for the `Set` abstraction have the following signatures:

```
difference(Set[T]): Set[T];
symmetric_difference(Set[T]): Set[T]
```

and the associated constraints are:

```
S1.difference(S2).element(X) :- S1.element(X),
                                ~S2.element(X);
S1.symmetric_difference(S2).element(X) :- S1.element(X),
                                          ~S2.element(X);
S1.symmetric_difference(S2).element(X) :- S2.element(X),
                                          ~S1.element(X)
```

As can be observed, the above constraints involve non-stratified negation.

The semantics of non-stratified logical clauses is captured by means of a partial truth assignment on the Herbrand base. The *well-founded* semantics [42] of logic programs (with no restriction on negation) is a partial assignment of truth values obtained by two fixpoint operators, one to derive positive facts and the other to derive negative facts. The operator used to derive positive facts is the usual immediate-consequence operator. For the derivation of negative facts the notion of *unfounded sets* is required. Informally, an unfounded set A with respect to a partial interpretation I is a set of atoms of the Herbrand base of a logic program P which satisfy the following property:

> $p \in A$ iff for each ground instance of a clause whose head is p, either (1) some subgoal of the clause is inconsistent with I or (2) some positive subgoal occurs in A.

Intuitively, I is regarded as what is already known about the intended model of P. Condition (1) says that the rule instance cannot be used to derive p. Condition (2), referred to as the *unfoundedness condition*, states that of all the rules that still might be usable to derive some atom in set A, each requires an atom in A to be true. In other words, there is no first atom in A which can be established to be true. Consequently, all of the atoms in A are assumed to be false in the well-founded semantics. Of course, this process has to be iteratively performed to arrive at the final well-founded model.

As far as the unification process is concerned, we still use E-unification described earlier, however, we need to provide a mechanism for solving negated subgoals. We use the following technique proposed by Clark [14] called negation as finite failure to resolve negated subgoals.

Consider a set of clauses P and the goal $G = L_1, \ldots, L_n$. Consider a negated subgoal L_i, say $\neg A$. If $P \cup \{\leftarrow A_i\}$ has a finitely failed resolution tree then,

$$\leftarrow L_1, \ldots, L_{i-1}, L_{i+1}, \ldots, L_n$$

is a resolvent of G.

With this modification (for negated subgoals), we can use the derivations defined in an earlier section to arrive at solutions to the queries.

11.4 Multiple Inheritance

The usual problems related to the complexity of multiple inheritance seem to have an interesting relationship with the complexity of the required object-oriented logic-based paradigm. For example, if we try to derive Set by multiple inheritance from SimpleSet and Bag, we get the following specification.

```
Specification Set[T];
Inherits SimpleSet[T] Select element, insert, delete;
                      Redefines element;
Inherits Bag[T] Select union, intersection, difference;
Constraints
   S.insert(X).element(X);
   S1.union(S2).element(X)  :- S1.element(X);
   S1.union(S2).element(X)  :- S2.element(X);
   S1.intersect(S2).element(X)  :- S1.element(X), S2.element(X);
   S1.difference(S2).element(X)  :- S1.element(X), ~S2.element(X);
End Set.
```

In the above specification, we used selective inheritance where selection of a particular method implies inheritance of the associated constraints and equational axioms (unless redefined). It is interesting that in this case, we have non-stratified negation in the body of the clauses. This is just an illustration of the problem we are investigating at present.

11.5 Disjunctive Clauses

There is at least one situation where disjunctive clauses (clauses with a disjunction in the head) would be required. The following specification of LinearOrder has a constraint whose head is a disjunction.

```
Specification LinearOrder[T <: Order[T]];
Constraints
   X.less_than(Y) or Y.less_than(Z);
   X.less_than(Z)  :- X.less_than(Y), Y.less_than(Z);
End LinearOrder.
```

Here, we do see a constraint that is a disjunctive fact. Computing with disjunctive specifications in general is not computationally efficient [1, 23], although there are several special cases of disjunctive clauses for which efficient algorithms do exist [17, 25]. We are currently exploring the possibilities of using some of these algorithms in our implementation.

12 Conclusions and Comparisons with Related Work

The paper presents a design and implementation of an object-oriented language with semantic constraints expressed in an extended logic programming paradigm.

The language is object-oriented, offering encapsulation and information hiding. It differs significantly from the existing strongly typed object-oriented languages in that it is declarative, rather than procedural. In addition to supporting strong and mostly static typing, the language enables software reusereuse via inheritance, subtype and parametric polymorphism. It features a semantic specification facility based on a suitably extended logic programming framework reflecting the behavioral nature of the object-oriented paradigm. It provides higher-order polymorphic features proposed in advanced object-oriented type systems, in spite of the fact that the language is first-order.

Our main contribution is in enhancing a sophisticated object-oriented type system with semantic specification facilities based on a suitable extension of the logic programming paradigm. In comparison with the object-oriented type systems, our semantic constraint language is a major generalization, and accomplishes behavioral subtyping mechanisms like those proposed in [5, 27, 28].

The underlying object-oriented type system is different from the type systems of languages such as C++ [40], or Eiffel [34]. C++ only recently supports parametric polymorphism; distinction between inheritance and subtyping is unclear in C++, and matching the two in Eiffel creates well-known problems [15]. Our paradigm is strictly more powerful and supports a semantic generalization of F-bounded polymorphism [11].

The type system of our language is comparable to [35]. However, the constraint language in [35] is first-order predicate calculus, which is computationally much more complex and does not have an initial algebra semantics.

The main advantage of our language is that it departs dramatically from the procedurality of the existing object-oriented languages, retaining advantages such as encapsulation, information hiding, inheritance, and object-identity. C++ does not have any constraint language and the assertion language of Eiffel is very limited.

Our constraint language is comparable to the one in EQLOG [19]. The difference is that ours is object-oriented and offers an interesting approach for modeling object states and their transitions. In addition, our constraint language consists of pure Horn clauses with separate equality theory, whereas in EQLOG equality is merged with the Horn clauses. Our paradigm is thus not only more intuitive but also computationally more tractable. In addition, EQLOG does not handle negation, and our paradigm is capable of dealing with some forms of negation as explained in Section 11.

The language implementation technique extends some object-oriented and logic programming architectures by a novel, automata-based model for representing object states and state transitions. Our contribution is a novel way of representing states as equivalence classes of messages. As such the model resembles the related approaches in algebraic automata theory.

The language and its initial implementation are currently used as a powerful prototyping tool, even if a production quality object-oriented language is used for the actual implementation.

The following are some possible directions (a few of which are quite ambitious) for future work:

- Investigate the problems associated with multiple inheritance in a truly behavioral and declarative paradigm.
- Extend the paradigm with persistence and develop an appropriate underlying system's architecture.
- Extend the language with appropriate high-level action composition rules.
- Provide a semantic definition for the language in a suitable formal framework.
- Develop sophisticated optimization techniques for automatic generation of the procedural implementations of methods from their specifications.
- Increase the expressive power of the constraint language by allowing advanced features such as negation and equality in the clauses, while maintaining soundness by ensuring that some semantic interpretation (preferably the initial one) is still available.

References

1. S. Abiteboul, P. Kanellakis, and G. Grahne. On the representation and querying of sets of possible worlds. In *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, pages 34–48, 1987.
2. S. Alagić. *Object-Oriented Database Programming.* Springer-Verlag, New York, 1989.
3. S. Alagić. Generic modules, kinds and polymorphism for Modula-2. In *Proceedings of the 2nd International Modula-2 Conference*, Loughborough, England, 1991.
4. S. Alagić. Persistent meta-objects. In A. Dearle, G.M. Shaw, and S.B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice*, pages 31–42. Morgan Kaufmann, 1991.
5. P. America. Designing an object-oriented programming language with behaviorial subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands*, pages 60–90. Lecture Notes in Computer Science 489, Springer-Verlag, 1991.
6. K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B (Formal Models and Semantics).* MIT Press, 1990.
7. K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, Los Altos, 1988.
8. R. Bagai, M. Bezem, and M. H. van Emden. On downward closure ordinals of logic programs. *Fundamenta Informaticae*, XIII(1), March 1990.

9. K. Bruce. Safe type checking in a statically typed object-oriented programming language. In *Proceedings of POPL (Principles of Programming Languages)*, pages 285–298, 1993.
10. P. Canning, W. Cook, W. Hill, and W. Olthoff. Interfaces for strongly typed object-oriented programming. In *Proceedings of the OOPSLA Conference*, pages 457–467, 1989.
11. P. Canning, W. Cook, W. Hill, W. Olthoff, and J.C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Conference on Functional Programming and Computer Architecture*, pages 273–280, 1989.
12. L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.
13. L. Cardelli, J. Donahue, M. Jordan, B. Kalslow, and G. Nelson. The Modula-3 type system. In *Conference Record, ACM Symposium on Principles of Programming Languages*, pages 202–212, 1989.
14. K. L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases.* Plenum Press, NY, 1978.
15. W. Cook. A proposal for making Eiffel type safe. *The Computer Journal*, 32(4):305–311, 1989.
16. W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings of Principles of Programming Languages*, pages 125–135. ACM Press, 1990.
17. M. Dalal. Some tractable classes of disjunctive logic programs. Technical Report, Department of Computer Science, Rutgers University, 1992.
18. M. Fay. First-order unification in an equational theory. In *Proceedings of the 4th Workshop on Automated Deduction*, pages 161–167, 1979.
19. J. Goguen and J. Meseguer. EQLOG: equality, types and generic modules for logic programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 295–363. Prentice Hall, 1986.
20. J. Goguen and J. Meseguer. Unifying functional, object-oriented and relational programming with logical semantics. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT Press, 1987.
21. P. Grogono. Issues in the design of an object-oriented programming language. *Structured Programming*, 12:1015, 1991.
22. J.-M. Hullot. Canonical forms and unification. In W. Bibel and R. Kowalski, editors, *Proceedings of the 5th Conference on Automated Deduction*, pages 318–334. Lecture Notes in Computer Science 87, Springer-Verlag, 1980.
23. T. Imielinski. Incomplete deductive databases. *Annals of Mathematics and Artificial Intelligence*, 3(2-4):259–293, 1991.
24. J. Jaffar, J.-L. Lassez, and M. J. Maher. A logic programming language scheme. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 441–468. Prentice Hall, 1986.
25. Z. A. Khandaker, J. A. Fernandez, and J. Minker. A tractable class of dis-

25. Z. A. Khandaker, J. A. Fernandez, and J. Minker. A tractable class of disjunctive deductive databases. In *Proceedings of the Joint International Conference and Symposium on Logic Programming, Washington D.C.,*, 1992.
26. W. A. Kornfield. Equality for Prolog. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 279–294. Prentice Hall, 1986.
27. G. Leavens and W.E. Wiehl. Reasoning about object-oriented programs that use subtypes. In *Proceedings of ECOOP/OOPSLA-90*, pages 212–223, 1990.
28. B. Liskov and J.M. Wing. A new definition of the subtype relation. In *Proceedings of ECOOP-93*, 1993.
29. J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.
30. D.B. MacQueen. An implementation of standard ml modules. In *Proceedings of the ACM Conference on LISP and Functional Programming*, pages 212–243, 1988.
31. J. McLean. A formal method for abstract specification of software. *Journal of the ACM*, 31(3):600–627, 1984.
32. J. Meseguer and X. Qian. A logical semantics for object-oriented databases. In *Proceedings of ACM-SIGMOD Conference*, 1993.
33. B. Meyer. *Object-oriented Software Construction.* Prentice Hall, 1988.
34. B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.
35. J. Mitchell, S. Meldaland, and N. Madhav. An extension of standard ML modules with subtyping and inheritance. In *Proceedings of Principles of Programming Languages*, pages 270–278, 1990.
36. A. Ohori and P. Buneman. Static type inference for parametric classes. In *Proceedings of OOPSLA'89*, pages 445–456, 1989.
37. J. Palsberg and M.I. Schwartzbach. Object-oriented type inference. In *Proceedings of OOPSLA'91*, pages 146–161, 1991.
38. G.D. Plotkin. Building in equational theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence, 7*, pages 73–90. Halsted Press, 1972.
39. A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA-86*, pages 38–45, 1986.
40. B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1993.
41. R. Sunderraman. Deductive databases with conditional facts. *Lecture Notes in Computer Science*, 696:162–175, 1993.
42. A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):621–650, 1991.