Transactions on Information and Communications Technologies vol 9, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

A prototyping technology for typed object-oriented software development

R. Bagai, S. Alagić & R. Sunderraman
Department of Computer Science, Wichita State University, Wichita, KS 67260-0083, USA

Abstract

We present a typed object-oriented prototyping technology and its associated methodology for large-scale quality software development. The technology supports a design methodology that leads to the development of strongly and largely statically typed software. In addition to pushing static type checking to its limits, the proposed technology relies heavily on Horn logic based specifications that capture the semantic subtleties of the target paradigm. The underlying type system supports advanced features, such as inheritance, behavioural subtyping, and subtype and parametric polymorphism. The core of the technology is an object-oriented, logic based, strongly-typed specification language, and its interpretive implementation. The result is a powerful prototyping tool for software development, which permits rapid prototyping of complex object-oriented design decisions. Such a tool can be used in association with a programming language (such as Eiffel or C++), prior to getting into the actual coding phase, to enable design correctness verification. We demonstrate the power of the tool by presenting a number of fundamental software abstractions written in our language. We also outline an implementation of the language.

1 Introduction

A highly desirable precondition for initiating any large-scale project is an acceptable degree of confidence in the project's ability to meet the intended goals. Development of large software systems is no exception. Timely verification of their specifications for correctness, adequacy and feasibility very often results in a higher quality product and avoids much human effort in the later stages of the software's life-cycle.

In this paper we present a rapid prototyping technology and its associated methodology for typed object-oriented software systems. The technology provides for quick construction of prototypes for strongly and mostly statically typed object-oriented systems. The prototypes thus constructed can be verified before the actual production quality systems are developed.

The object-oriented paradigm is primarily intended to better manage complex software systems by providing mechanisms for software reuse, information hiding, and the like. Thus, prototyping design decisions involving classes, inheritance hierarchies, subtype and parametric polymorphism, becomes our major design concern. Providing these in a single language, while pushing static type checking to its limits is a challenge, for which current research has not yet succeeded in producing acceptable solutions. Some results have been reported in Cook [6], Cook *et. al.* [7], and Snyder [17].

A prime motivation driving the evolution of programming languages in general has been to eliminate the gap between the specifications and implementations of software systems. While each step in the evolution, such as from machine languages to assembly languages to high-level languages, did contribute toward the goal, it was only with the advent of declarative languages that the goal seemed *almost* within grasp. This is because programs in declarative languages are executable specifications. Currently, a major drawback of these languages is the lack of efficient implementations for them. However, for prototyping purposes, where efficiency is not a major concern, declarative languages prove to be most suitable.

We adopt the logic programming paradigm for constructing the prototypes of the object-oriented systems. Unfortunately, pure Horn clause logic, or even predicate calculus, is not expressive enough for specifying most classes. Some form of equational theory is necessary.

The proposed prototyping technology integrates object-oriented design with the logic programming paradigm, into which a separate equality theory is incorporated. The underlying language is strongly and mostly statically typed, and supports software reuse techniques, such as inheritance, subtype and parametric polymorphism. It allows behavioural subtyping (see Liskov and Wing [12]), in addition to the conventional syntactic one. Moreover, it offers higher-order polymorphic facilities (such as F-bounded polymorphism, Canning *et. al.* [3]) in a first-order framework.

The rest of this paper is organised as follows. Section 2 presents the salient features of the prototyping language. Section 3 outlines how some advanced object-oriented features, such as behavioural subtyping, F-bounded polymorphism are provided. Section 4 describes an implementation mechanism based on automata theory. Finally, Section 5 gives some concluding remarks, comparisons with related work and directions for future work.

Transactions on Information and Communications Technologies vol 9, © 1994 WIT Press, www.witpress.com, ISSN 1743-3517

2 The Prototyping Language

The basic components of our prototyping language are that of most object-oriented languages, namely objects, classes and methods. However, the language is strongly typed, polymorphic and the method definitions in it are declarative in nature. Its main construct is the specification block, which provides for encapsulation and information hiding. A specification block specifies an object class (also called a *type*), and consists of the following components:

- A collection of optional type parameters. If present, the underlying specification becomes polymorphic.
- A collection of creators. When invoked, creators return newly created objects of that class. Different funtors of the same class differ from each other in the initial state of the objects they return.
- A collection of mutator methods. These are methods that may affect the state of the underlying object, while preserving the object identity.
- A collection of function methods. These are methods that preserve both state as well as the identity of the underlying object. They return values of that class.
- A collection of predicate methods. Like function methods, these methods also preserve both state as well as the identity of the underlying object. However, they return only truth values (TRUE or FALSE).
- A collection of constraints. These are essentially axioms that define the predicate methods in terms of the mutator and function methods of that class.
- A collection of equality axioms. The equality axioms of a class form an equality theory on the set of all mutator application sequences on objects of that class.

Let us consider an example prototype of a class of bags of objects of class `T`. (Like sets, bags are unordered collections, but may have multiple memberships of objects in them.) The following specification relies on the specification of the class `Nat` of natural number objects. We assume that `Nat` is equipped with functions `succ` for successor and `add` for addition.

```
Specification Bag[T];
Imports Nat;
Creators
  null;
```

```
Mutators
  insert(T);
  delete(T);
Functions
  union(Bag[T]);
Predicates
  non_empty();
  member(T, Nat);
Constraints
  B.non_empty() :- B.member(X, N.succ());

  null.member(X, 0);
  B.insert(X).member(X, N.succ()) :- B.member(X, N);
  B1.union(B2).member(X, M.add(N)) :-
                B1.member(X, M), B2.member(X, N);
Equality
  B.insert(X).delete(X) = B;
  B.insert(X).insert(Y) = B.insert(Y).insert(X);
End Bag.
```

The above is a specification of a polymorphic class `Bag` with a parameter type `T`. The specification uses the class `Nat` of natural numbers; hence the `Imports` clause. The class `Bag` has a creator `null` that, intuitively speaking, returns a newly created empty bag upon invocation. An object of class `Bag` is equipped with mutator methods `insert` and `delete`, which perform the fundamental element insertion and deletion operations on that object. The `union` is a function method that does not affect the state of the underlying object, and returns a new bag object. The predicate method `non_empty` returns TRUE if the underlying bag is not empty. The predicate method `member(X, N)` returns TRUE if the object `X` belongs `N` times to the underlying object.

The `Constraints` section of the specification contains axioms that define all predicates in terms of mutator and function methods. For this example, the language of Horn logic is sufficient. We adopt this logic because it is easy to understand, interpret, and has well-known completeness results (see Apt [2] for an exposition). While Horn logic suffices for many applications, more complex classes require more expressive languages (such as Normal Horn clauses or Disjunctive Horn clauses), for which implementations can be constructed.

Observe that no direct specification is given for the mutator and function methods. Their results are observed only through the predicate methods. Thus, a set of axioms for a class is complete if each predicate method is defined (TRUE or FALSE) for each mutator/function sequence. If a default rule for negation, such as the Closed World Assumption (Lloyd [13]),

is adopted, then a predicate is assumed to be false for any mutator/function sequence for which it is not defined.

The `Equality` section in the above `Bag` specification states that the order of insertions of an element into a bag is not important, and that the deletion of an object in a bag undoes one of that object's past insertions. Any such user-supplied equality theory needs to be respected by the underlying unification mechanism of any implementation.

Any mutator sequence involving invocations of the `delete` method can be seen to be equivalent (with respect to the equality theory) to one without it. We call such methods *vanishable*. In order for the axioms in the `Constraints` section to be complete, all predicate methods should be defined for all the non-vanishable function and mutator methods. For this reason, the predicate `member` in the above specification of `Bag` is not defined for any method invocation sequence involving `delete`.

An example usage of the above specification might look like the following:

```
let GroceryBag1 = null[GroceryType];
let GroceryBag2 = null[GroceryType];
...
let AllGroceries = GroceryBag1.union(GroceryBag2);
```

Intuitively, the bag `AllGroceries` now contains all the groceries contained in the bags `GroceryBag1` and `GroceryBag2`.

Inheritance

From a specification of a class C_1 the language permits deriving another class C_2 that inherits all the creators, methods (possibly after renaming), constraints and equality axioms of C_1. Any inherited methods may be redefined in C_2. In addition, class C_2 may introduce more methods, together with their constraints and equality axioms. This provides for the usual syntactic notion of subtyping (Cardelli and Wegner [5]). A more semantic notion of subtyping, called *behavioural subtyping* (Liskov and Wing [12]) is described in the next section.

As an example, consider the following specification of the class of all sets of objects of class `T`. Sets differ from bags in that they may contain any element only once.

```
Specification Set[T];
Inherits Bag[T];
Predicates
  element(T);
```

```
    member(T, Nat) redefines member;
  Constraints
    S.element(X) :- S.member(X, 1);

    S.insert(X).member(X, 1);
    S.delete(X).member(X, 0);
    S1.union(S2).member(X, 1) :- S1.member(X, 1);
    S1.union(S2).member(X, 1) :- S2.member(X, 1);
  End Set.
```

The term `1` used in the above specification is considered to be equated to the term `0.succ()` in the specification of `Nat`. The `Set` specification redefines the predicate method `member` of the parent class `Bag` and the new definition of `member` is applicable to all sets. A new predicate method `element` is introduced and defined. However, the definition of `non_empty` and the equality axioms are inherited from the `Bag` class as is.

3 Advanced Object-Oriented Features

In this section we demonstrate two advanced object-oriented features of our language, namely behavioural subtyping (Liskov and Wing [12]) and higher-order polymorphism (Canning *et. al.* [3]), and exhibit their usefulness in constructing prototypes of many well known object-oriented classes.

Behavioural Subtyping

Although the class `Set` defined in the previous section inherits from the class `Bag`, the inheritance does not obey the *principle of substitutability* associated with subtypes, which is that an object of a subtype may be used anywhere its supertype is legal. This is easily seen by attempting to insert the same object more than once in a set. The inserted object can belong only once to the set, and a single deletion can remove it from the set. Such behaviour of a set is unlike that of a bag.

In many applications a stronger notion of subtyping is desirable, in which an object of a subtype always behaves like its supertype. Liskov and Wing [12] gave necessary conditions for the methods of the subtype in order for it to be a *behavioural subtype* of its supertype. The subtype should essentially provide all methods of its supertype (after possible renamings) with compatible signatures. The pre/post-conditions of the subtype methods should follow the contra/covariance rules of Cardelli [4]. Moreover, any new methods introduced by the subtype should be completely describable in terms of existing methods. Under these conditions, an object of the subtype will never exhibit behaviour that cannot be expected from objects of the supertype.

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

```
Specification Stack[T];
Creators
  null;
Mutators
  push(T);
  pop();
Predicates
  top(T);
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. Since `pop` is a vanishable mutator, `top` is defined only for mutator invocation sequences involving `push`.

Let us now consider the following list specification:

```
Specification List[T];
Inherits Stack[T];
Creators
  nil renames null;
Mutators
  cons(T) renames push;
  cdr() renames pop;
  concat(List[T]);
Predicates
  car(T) renames top;
  is_nil();
  length(Nat);
Constraints
  L.is_nil() :- L.length(0);

  nil.length(0);
  L.cons(X).length(N.succ()) :- L.length(N);
Equality
  nil.concat(L) = L;
  L1.cons(X).concat(L2) = L1.concat(L2).cons(X);
End List.
```

The class `List` inherits all creators, methods, constraints and equality axioms from the class `Stack`. However, the creator `null` is renamed to `nil`. Similarly, the methods `push`, `pop` and `top` are renamed to `cons`, `cdr` and `car`, respectively. An important point to observe is that the new mutator `concat` introduced in `List` specification is vanishable. The equality axioms for it are contained therein. Since the predicates `is_nil` and `length` are also specific to lists, the specification contains clauses for them.

Higher-order Polymorphism

The main language construct, the specification block, comes in general with a type parameter. That feature 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 $\rightarrow$ 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 `Nat`, `Real`, `String` etc. which are in fact predefined and thus have fixed interpretations. Standard definitions for basic abstractions such as `List`, `Tree`, `Set` etc. are also provided, but a user can always redefine those using the facilities of the language.

A particularly important form of higher-order polymorphism for object-oriented languages is F-bounded polymorphism (Canning *et. al.* [3]). 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 binary relation. Let us call the relation `less_than`. Its generic (parametric) definition is given below.

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

A specific ordering will now be specified by introducing appropriate constraints. For example, a partial order is just transitive and anti-symmetric, as specified below. But the type parameter `T` of the partial order abstraction must be a subtype of `Order[T]`, denoted `T <: Order[T]`. This condition guarantees that whatever the underlying type `T` is, it must have the predicate `less_than`. This form of subtyping is not just bounded by a particular specification. Rather, an F-bound is `Order[T]`.

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

```
Equality
  X = Y :- X.less_than(Y), Y.less_than(Z);
End Order.
```

Although F-bounded polymorphism 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, only when the F-bounded subtyping condition is satisfied the universal quantification of all variables (standard in logic programming and adopted in our language design) makes sense.

For a practical example of this feature, consider the following specification of the class of sortable lists. The specification depends upon the type `LinearOrder`, which can be an appropriately defined subtype of `PartialOrder` defined above.

```
Specification SortableList[T <: LinearOrder[T]];
Inherits List[T];
Mutators
  sort();
Predicates
  sorted();
Constraints
  nil.sorted();
  nil.cons(X).sorted();
  L.cons(X).cons(X).sorted() :- L.cons(X).sorted();
  L.cons(X).cons(Y).sorted() :- Y.less_than(X),
                                L.cons(X).sorted();
Equality
  L.sort() = L :- sorted(L);

  L1.concat(L2.cons(X).cons(Y)).sort() =
  L1.concat(L2.cons(Y).cons(X)).sort()
                                :- X.less_than(Y);
End List.
```

The above specification guarantees that the type parameter `T` is equipped with the predicate `less_than` that satisfies the axioms for a linear order. The mutator `sort` results in sorting the underlying list and the predicate `sorted` indicates whether or not the underlying list is sorted.

The following is another example of a specification made possible by F-bounded polymorphism. It specifies the class of array objects, parameterized by the index and element types. Its creator `new_array` is invoked with the lower and upper bounds of the array.

```
Specification Array[B <: LinearOrder[B], E];
Creators
  new_array(B, B);
Mutators
  put(B, E);
Predicates
  lower(B);
  upper(B);
  get(B, E);
Constraints
  new_array(L, U).lower(L);
  A.put(I, X).lower(L) :- A.lower(L);

  new_array(L, U).upper(U);
  A.put(I, X).upper(U) :- A.upper(U);

  A.put(I, X).get(I, X) :-
              A.lower(L), A.upper(U),
              less_equal(L, I), less_equal(I, U);
Equality
  A.put(I, X).put(J, Y) = A.put(J, Y).put(I, X)
                                    :- not(I = J);
End Array.
```

The equality axiom in the above specification involves negation, which for arbitrary predicates is quite difficult to implement. However, when used only for disequality, it is surprisingly easy to provide (see Goguen and Meseguer [9]). If an attempt to unify the two terms involved fails, the disequality holds.

4 Implementation

The sophistication level of a mechanism capable of 'executing' the user supplied specifications clearly depends on that of the language used to express their constraints. For the `Bag` and `Set` examples presented earlier, and for many other non-trivial applications, the language of Horn clause logic with unconditional equality axioms is sufficient. While there exist applications for which the underlying language needs to be more expressive (such as full first order predicate calculus with equality), for now we limit ourselves to the logic of Horn clauses with unconditional equality, as it is known to be computationally tractable (Goguen and Meseguer [9]). Any such set of clauses is known to possess a unique initial model, up to isomorphism, for which there exists a complete inference mechanism based on universal unification that respects the underlying equality theory (Plotkin [16]). The situation is

somewhat similar when the equality theory itself consists of Horn clauses, which are studied in Jaffar *et. al.* [11].

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.

Translation

Associated with each object class in a given program is a collection of its predicate, function 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.

Similarly, for any n-ary function or mutator method f belonging to a class c, we introduce an $(n+1)$-ary function symbol f_c. That we can treat function and mutator methods alike becomes clear when we describe object states later.

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

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

in the class `Bag` now becomes

```
member_Bag(insert_Bag(B, X), X, succ_Nat(N)) :-
        member_Bag(B, X, N);
```

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

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 (function and mutator invocations) each one beginning with a creator application. This model is in fact based on a technique from algebraic automata theory.

For example, if `x` is a variable of type `Bag`, the action

```
let x = null[Nat];
```

creates a bag of natural numbers, whose current state is the term `null_Bag`, and due to the axiom involving the `null` creator, 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_Nat), 1_Nat)
```

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

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

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

```
z.member(0, N)
```

produces the solution

```
N = 0.succ().succ()
```

indicating that the element `0` belongs 2 times to the object `z`.

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 Goguen and Meseguer [9]. Terms t_1 and t_2 are *equivalent* iff $t_1 = t_2$ is provable from the given set E of equality axioms.

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

$$b_1 = \texttt{null.insert(0).insert(1)}$$
$$\rightarrow \texttt{null.insert(1).insert(0)}$$

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

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 unification algorithm proposed in Fay [8] and improved in Hullot [10] is known to be complete. It produces a complete set of unifiers, i.e. for a given set E of rewrite rules and terms t_1 and t_2, 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, $\gamma = \theta\delta$.

Role of Deductive Capabilities

As is evident in our implementation technique, mutator and function methods are never really 'executed'. Thus, the constraints in the prototypes need not contain any information for doing that. The constraints, however, should contain enough information about the *effects* of the mutator and function methods on the predicate methods, since the predicate methods provide the only ways for observing the underlying objects.

For instance, in the `Bag` example the bag object returned by the method `union` is not at all described. In particular, the specification does not entail

that `union` is, say, commutative or associative. Moreover, at the level of logic terms, the specified `union` method does not hold these properties. 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. Then the state of the bag `b12` after the action

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

is

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

whereas the state of the bag `b21` after the action

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

is

```
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 predicates contained in the `Bag` specification.

5 Conclusions and Related Work

We have presented a technology for constructing prototypes of typed object-oriented software systems. The core of the technology is a logic based, strongly-typed specification language, and its interpretive implementation. The prototyping language integrates object-oriented design with an extended logic programming paradigm. In a declarative setting, the language offers encapsulation and information hiding, and provides software reuse mechanisms like inheritance, and subtype and parametric polymorphism. As an implementation, we presented a novel way of representing states as equivalence classes of messages. As such the model resembles the related approaches in algebraic automata theory. The underlying architecture is that of logic programming, in which unification is performed with respect to an equality theory.

The underlying object-oriented type system of our language is different from the type systems of languages such as C++ (Stroustrup [18]), or Eiffel (Meyer [14]). 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 (see Cook [6]). Our paradigm is strictly more powerful and supports a semantic generalization of F-bounded polymorphism (Canning *et. al.* [3]).

The type system of our language is comparable to the one in Mitchell *et. al.* [15]. However, the constraint language therein is first-order predicate calculus, which is computationally much more complex and does not have an initial algebra semantics.

Our constraint language is comparable to the one in EQLOG (Goguen and Meseguer [9]). 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 comparison with the object-oriented type systems, our semantic constraint language is a major generalization, and accomplishes behavioral subtyping mechanisms like those proposed in Liskov and Wing [12] and America [1].

Although a paradigm in which constraints are expressed in Horn logic with separate equality axioms is simple to understand and computationally tractable, there are situations that need additional features such as equality in the clauses, negation in the body of clauses and, possibly, disjunctions in the head of the clauses. We leave such powerful and useful extensions of our paradigm for future work.

References

[1] P. America. Designing and object-oriented programming language with behaviorial subtyping. In *Foundations of Object-Oriented Languages, REX School/Workshop, Noordwijkerhout, The Netherlands*, pages 60–90. LNCS 489, Springer-Verlag, 1991.

[2] K. R. Apt. Logic programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B (Formal Models and Semantics)*. MIT Press, 1990.

[3] 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.

[4] L. Cardelli. A semantics of multiple inheritance. *Information and Computation*, 76:138–164, 1988.

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

[6] W. Cook. A proposal for making Eiffel type safe. *The Computer Journal*, 32(4):305–311, 1989.

[7] W. Cook, W. Hill, and P. Canning. Inheritance is not subtyping. In *Proceedings of Principles of Programming Languages*, pages 125–135. ACM Press, 1990.

[8] M. Fay. First-order unification in an equational theory. In *Proceedings of the 4th Workshop on Automated Deduction*, pages 161–167, 1979.

[9] 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.

[10] 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. LNCS 87, Springer-Verlag, 1980.

[11] 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.

[12] B. Liskov and J.M. Wing. A new definition of the subtype relation. In *Proceedings of ECOOP-93*, 1993.

[13] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, second edition, 1987.

[14] B. Meyer. *Eiffel: The Language.* Prentice Hall, 1992.

[15] 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.

[16] G.D. Plotkin. Building in equational theories. In B. Meltzer and D. Michie, editors, *Machine Intelligence, 7*, pages 73–90. Halsted Press, 1972.

[17] A. Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Proceedings of OOPSLA-86*, pages 38–45, 1986.

[18] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, second edition, 1993.