Formal Aspects of Computing (2002) 13: 214–232
© 2002 BCS

Formal Aspects of Computing

Should ML be Object-Oriented?

David MacQueen
University of Chicago, Chicago, Illinois, USA

Abstract. At a fundamental level, functional and object-oriented programming languages are all 'higher-order', in the sense that they support computing with values that are themselves pieces of program code encapsulated with a local environment. In functional languages these 'active' values are functions, while in object-oriented languages they are objects. Both styles of higher-order language claim to provide good support for writing adaptable programs, but functional and object-oriented languages achieve this adaptability in different ways: functional programs rely on parameterisation at the value, type and module level, while object-oriented languages rely primarily on subtyping and implementation inheritance. Here we compare these two approaches, mainly in terms of the features and properties of their type systems, and consider the benefits and disadvantages of unifying (or merging) the two paradigms by adding object-oriented features to ML as a base language. We argue that while some of the simpler aspects of object-oriented languages are compatible with ML, adding a full-fledged class-based object system to ML leads to an excessively complex type system and relatively little expressive gain, especially if we aim to preserve that mostly functional style of programming that is a major advantage of ML.

Keywords: Functional programming; Inheritance; ML; Modules; Object-oriented programming; Polymorphism; Subtypes

1. Introduction

The sophisticated type systems and higher-order features of HOT (higher-order typed) languages give them the expressive power needed for building program components and libraries that are easy to adapt for reuse in different contexts. However, the techniques used by functional and object-oriented languages to achieve expressive power and adaptability are markedly different: functional languages rely mainly on various forms of *parameterisation* for flexibility, while object-oriented languages employ *subtyping* and *open recursion* (in the form of implementation inheritance or delegation).

The last twenty years has seen substantial development of the foundations of functional languages and their type systems, based on pioneering work by Landin [Lan66], Burstall [Bur69], Reynolds [Rey74] and Milner [Mil78] in the 1960s and 1970s. A number of fully developed, practical language designs on the functional side (e.g. ML and Haskell), have been based on these foundations. In the development of object-oriented languages, an empirical tradition long dominated (Simula, Smalltalk, C++), but since Cardelli's

Correspondence and offprint requests to: David MacQueen, University of Chicago, Chicago, IL 60637, USA. Email: macqueen@cs.uchicago.edu

1984 paper [Car84], the semantic and type theoretic techniques developed for functional languages have been applied extensively to the study of object-oriented languages, and some prototype language designs have derived from this theoretical work (e.g. [BFP97]).

Recently, there has been much research and design activity inspired the idea of combining functional and object-oriented features in a single language. This became the central issue in discussions of possible next-generation versions of the ML language [MWG99], and several concrete proposals have emerged that address this challenge, such as OCaml [Ler96] and its precursor, ML-ART [Rem94a], Object ML [ReR96a, ReR96b], and Moby [FiR99].

This paper considers the problem of the synthesis of functional and object-oriented programming specifically in terms of adding features to Standard ML (SML) to give it more object-oriented capabilities. We regard SML as a very successful, but not perfect, language design, and ask whether it can be improved by borrowing features and ideas from object-oriented programming. We argue that adding a full, class-based object system would not improve the language, but some basic ideas like subtyping could be adapted to SML and would enhance its power and flexibility without damaging its nature and effectiveness. In the course of making this argument, we will compare the basic elements of the two paradigms to see where they overlap and where they diverge. Our goal here is not new technical results, but a critical evaluation of language design options.

We take Standard ML as our representative functional language for several reasons: (1) it is a well-defined and fairly pure instance of the ML family of languages; (2) as a call-by-value language with side effects, it is closer to object-oriented languages than a pure, call-by-need functional language, and (3) we have an interest in the future evolution of the ML family starting from Standard ML.

A historical note. At the core level, ML is characterised by polymorphic types with Hindley–Milner type inference [Mil78] and by algebraic datatypes, first described by Burstall [Bur69] and first implemented as part of a statically typed language by Burstall, MacQueen and Sannella in the language Hope [BMS80]. At the module level, ML provides interfaces (*signatures*), modules (*structures*) and parametric modules (*functors*) whose design was directly inspired by the parametric algebraic specifications of Burstall and Goguen's Clear specification language [BuG77, BuG80]; see also [Mac81]. Burstall was a principal designer of Hope, and he was also a primary contributor to the Standard ML design. It is clear that Burstall was at the centre of these developments from the 1960s through the 1980s, and he has had a profound influence on many of the participants through his advocacy of language designs based on clear, simple and well-founded ideas.

1.1. Overview

We start by reviewing the core ideas of the functional paradigm, leading to a more detailed consideration of the features provided by Standard ML for constructing programs: function values, parametric polymorphism, datatypes and modules.

We then discuss the basic ideas of object-oriented programming and the embodiment of those ideas in language features such as objects, classes, subtyping and inheritance. Instead of using a particular existing object-oriented language (e.g. Java) as a basis for discussion, we will discuss object-oriented features in a more idealised context, and often in terms of how they can be encoded in a functional language like ML or $F^{\omega}_{<}$, a higher-order, explicitly typed lambda calculus enriched with subtyping, recursion, references and other features [CaW85, Car89, PiT94, HoP95, Pie02]. Dealing with an 'idealised' object-oriented language, with features derived from first principles and expressed in a functional framework, allows us to avoid accidental complications and outright errors of naive designs and makes it easier to see how OO ideas and features relate to a functional language framework.

Finally, we make some observations on the relative advantages and disadvantages of the FP and OO features, considering the trade-off between expressive power and flexibility versus feature and language complexity. We observe that the basic elements of ML have relatively straightforward semantics and behaviour, while the basic elements of object-oriented programming appear intrinsically subtle and difficult. Another point of comparison is the relative tractability of parameterisation in comparison with open recursion. We note the perceived need for 'modelling languages' like UML in the OO world, while on the other hand functional languages serve as their own modelling languages – indeed, (pure) functional programs are often treated as executable specifications.

We will assume some familiarity with Standard ML, with formal type systems [Car97, Pie02] and with

$F^{\omega}_{\leq}$. Good tutorial introductions to Standard ML are available from [Har], [HaR99] and [Pau96]. For a short tutorial on the SML module system, see [Tof96].[1] See also [CaW85] and [Car89] for descriptions of Fun and Quest, two fairly similar syntactically sugared languages based on $F^{\omega}_{\leq}$.

2. Functional Programming in ML

What basic benefits and capabilities does functional programming[2] offer? FP exploits a very simple framework that is powerful, flexible and robust, leading to programs that are easier to design, debug and reason about. The effectiveness of FP derives from the simplicity of the model together with the limited role of state and side effects.

The essential fact of FP is that functions are 'first-class' values. From this it follows that we can define functions that operate on functions or return functions as results (higher-order functions), and we can construct data structures with functions as components.

In addition, ML offers the flexibility of parametric polymorphism for implementing generic functions and data structures. As an additional benefit of the Hindley–Milner typing algorithm, we get automatic type reconstruction that relieves us of most of the burden of declaring types. The type system is simple and orthogonal, exploiting a small number of very general constructions (function spaces, products, tagged unions, recursion and parameterisation). It encourages the creation of new types and interfaces in support of the particular needs of an application.

The module system naturally generalises the core ideas and applies them to 'programming in the large'. Modules are hybrids combining types and values and they have their own types in the form of signatures. The module language is itself a simple functional language allowing us to write compound expressions to build programs from combinations of special-purpose and library modules.

Good programming style in ML is *mostly functional*, limiting the use of state and side effects and restricting them as far as possible to encapsulated abstractions. This style has been called *value-oriented* programming by Appel [App95]. In value-oriented programming, most computation is expressed in terms of constructing, analysing and transforming immutable values, which can be safely shared and reused without interference. As a consequence, logical errors deriving from side effects are minimised, and types have a prescriptive effect, meaning that types suggest or even determine behaviour to a far greater extent than in imperative programming. Another aspect of value-oriented programming is that value identity (for types where it is computable) is by default structural.[3]

Let us now take a more detailed look at the elements of SML programming and how they contribute to its power and adaptability.

2.1. Functions

We start with functions. Function values are self-contained closures combining the code that determines behaviour and an environment (a set of associated named values) that makes the function self-contained and provides resources and connections. They have a number of uses:

- Higher-order functions capture systematic patterns for using function parameters or creating function results. They can embody iterative control structures that capture common patterns of traversing data structures (e.g. the *map* and *fold* functions over lists).
- Higher-order functions are used to implement general control frameworks that are specialised by function parameters. Like inheritance in object-oriented programming, this allows generic, predefined code to call specialised, application-specific code (or put another way, old code can call new code). This also covers the notion of 'call-back' functions.

[1] Additional online tutorials and resources can be found at the SML/NJ web site: `http://www.smlnj.org/literature.html`.
[2] We will use FP as shorthand for functional programming.
[3] In lazy languages, we have the additional capability of working with potentially infinite data structures like streams, which are computed incrementally and only to the extent that they are needed. See [Hug89] for a good exposition of the advantages of this aspect of functional programming.

- Curried functions are another common form of higher-order function. A curried function can be 'partially applied' to a subset of its arguments as a way of creating a specialised variant of the function. For instance, a curried sort function taking a comparison operation and a list to be sorted is partially applied to a particular ordering to create a specialised sort.
- Functions can be embedded in data structures, allowing one to build active, dynamically evolving structures (e.g. streams). A collection of functions linked by a common environment can be built into a record to create a simple form of object, and if their common environment contains state the object encapsulates that state.
- Functions can be created with very little overhead, and can be anonymous. Function expressions can occur in any context of appropriate type. This encourages the creation of small, ad hoc functions, for instance as inline arguments to iteration functionals. Compare this with typical class-based OO languages, where in order to create an object one must first declare its class. This is particularly onerous if one intends to create a single, ad hoc object of the class.
- Functions can be defined or created within other functions. Such nested functions have access to the internal environment of their parent, and can be transmitted beyond the parent by being returned as a result or by being passed as an argument to an external function. This is analogous to the way that inner classes are used in Java, but involves much less syntactic overhead. An object generator function (a simple analogue of a class) is another example where nested functions escape via the object value. Nested functions are also useful for factoring and organising the code of a large function. For instance, in SML most loops are implemented by locally defining and calling a tail recursive function.
- Function abstraction can be used to delay or suspend evaluation (creating 'thunks'), and this technique is often used in control structures or active data structures (streams again).

2.2. Polymorphism

Parametric polymorphism is the second critical tool for writing flexible, generic code. Standard ML provides simple, prenex, predicative, polymorphic types based on the let-rule (the types of let-bound variables, including top-level declarations within modules, can be generalised). Polymorphism is an important complement to higher-order functions, allowing types to be specialised as the code is specialised by function parameters. The list *map* function is a typical example of this: without polymorphism, it would be very difficult to implement generic functions or interfaces.

The type reconstruction provided by the Hindley–Milner–Damas algorithm greatly increases convenience, at the cost of some acceptable limits on the expressive power of the type system. Good programming practice adds many type annotations that, while not strictly necessary, are useful both as documentation and as aids to improve error detection through redundancy.

2.3. Type Definitions

An essential component of program design in SML is the creation of new types appropriate to the task at hand. SML supports simple type abbreviations, datatypes and abstract types (normally implemented using opaque modules). Type abbreviations provide the convenience of more compact expression by naming recurring type expressions.

Datatypes are at the heart of the ML programming style. They combine tagged unions and recursion, and can be parametric (the list constructor exercises all three of these features). The variants of a datatype union are distinguished by *data constructors* (e.g. `nil` and `cons`), which are used to construct values and also to destruct and discriminate when used in patterns. Pattern matching over datatypes is a pervasive aspect of ML code.

Abstract types are opaque types (with hidden representation type) associated with an interface of values (constants, functions and exceptions) that are used to 'interpret' the abstract type.

All three forms of type definition can have parameters, in which case they define type operators that are used to build other types.

Type equality in ML is basically structural, except for datatypes and abstract types which use *nominal* type equality, i.e., each datatype or abstype declaration produces a statically unique type.

2.4. State

State is created in SML programs through the use of refs and arrays. The use of state and side effects provides a significant enrichment of the language at the cost of the possibility of errors caused by unintended interactions through side effects and aliasing. Potentially any function might have hidden side effects, and such side effects are usually not manifested in the types. It also becomes considerably harder to specify and reason about the behaviour of functions with side effects, and types are not as useful a hint about the behaviour of functions with side effects (e.g. consider a function of type `unit -> unit`).

On the other hand, the ability to create and update state makes it relatively easy to implement simple object-like structures in the form of records of functions sharing a common closure environment containing the object state.

The presence of state has an important impact on the type system, because stateful values cannot be polymorphic. The *value restriction*, adopted in Standard ML 97 to ensure soundness of the type system in the presence of state, slightly reduces the amount of polymorphism available, but the practical effect is minor. It is still quite possible to build generic functions or modules whose implementation uses state.

2.5. Exceptions

Exceptions are an essential tool used in ML for the usual purposes. They provide a disciplined way to deal with runtime failures, and they provide a control construct used for escaping from a deeply nested computation. The use of exceptions does not distinguish ML from object-oriented languages, except for the fact that exception values in ML are characteristically elements of a (special) exception datatype, while in OO languages they might be objects belonging to an exception class, as in Java.

2.6. Modules

Modules are collections of named components (a kind of generalised record) where the components may be types, values or nested modules. A key property of the module system is the independence of interface (signature) and implementation (structure or functor). A given signature may be implemented by multiple structures, while a given structure may implement (that is, it may match) multiple signatures. For instance, one may create two views of a given structure by matching it with two signatures, one of which might define a restricted public interface, while the other would provide a more revealing interface for the use of certain privileged modules.

One can also control information access by choosing between transparent and opaque signature matching. The latter reveals only interface information that is explicitly specified in the signature.

The ability of a single structure to match multiple signatures is related to the existence of a signature subtyping relation. A signature S_1 is a subsignature of S_2 if it has at least all the components of S_2, and if shared components have a more restrictive specification in S_1 (e.g. a polymorphic type in a value specification is more restrictive than one of its instances). This definition of the subsignature relation is a direct analogue of width and depth subtyping for record types.

Parametric modules, or *functors*, are used to define generic modules that can be instantiated, or specialised, but supplying appropriate modules as parameters. With functors, we have a simple typed functional language of modules (especially if *higher-order functors* are supported, as they are in Standard ML of New Jersey or Moscow ML). Using the module system to do functional programming 'in the large' allows us to flexibly compose programs from mixtures of generic and special-purpose modules.

The typing constructs and rules associated with the module system are a straightforward extension of those of the core language. The main novelties are signature matching and the signature subtyping it entails, type dependencies between module components (i.e. the fact that a type component name can appear in the specifications of later components), and the use of definitional specifications (or sharing declarations) to ensure necessary sharing of types among parameter modules.

3. Object-Oriented Programming

The task of describing the ‘essence’ of object-oriented programming is more challenging, because there are many different feature sets, languages and methodologies espoused by different camps within the OO community. One major division is between *class-based* and *object-based* languages [AbC96], Chapters 1 through 4), and to simplify our task we will restrict ourselves to the more popular and extensively studied class-based languages.

The appeal of OO is based on a naive metaphor of software ‘objects’ as models for real-world objects. The four principal attributes of class-based object-oriented programming are *encapsulation*, *inheritance*, *subtyping polymorphism* and *dynamic dispatch.*

- Objects *encapsulate* their internal state, allowing access to the state only through a set of functions, known as *methods*, associated with the object.[4] The set of methods (method suite) constitutes the external interface of the object, and an interface type for the object is made up of the types of the methods. Typically, the suite of interface functions is shared by objects of a given *class*, while the state is private to each object. The methods of an object are by default mutually recursive, and normally call each other indirectly through a variable (typically `self`) denoting the object of which they are members.
- *Implementation inheritance* refers to the ability to define new classes (called *subclasses*) by incrementally changing the functionality of an existing class. The incremental changes can take the form of additional state variables and methods, but they can also involve redefinition or *overriding* of old methods. Often the old (inherited) methods remain accessible through a special variable (typically `super`) even when overridden. Method overrides can rearrange the call-graph among the methods, affecting not only new methods, but the old, inherited methods as well. This effect is called *open recursion.*
- *Subtype polymorphism* is the typing basis for writing generic programs in classical OO languages. There is a natural structural subtyping on object interface types based on record subtyping, somewhat complicated by the fact that object interface types may be recursive. With subtyping comes the subsumption rule, which allows objects of a subtype to be used in any context that expects the supertype. Historically, object-oriented languages have associated (confused) subtyping with subclasses, but the soundness of such an identification requires restrictions on the flexibility of inheritance.
- The fourth major attribute is variously known as *dynamic dispatch*, *dynamic binding* or *polymorphism* (not to be confused with the parametric polymorphism of ML). This refers to the fact that a given object type may contain objects with different implementations for the same method, so when invoking a method of an object, the code that is executed is determined dynamically, by the value of the particular object, rather than statically as in a first-order, procedural language. Variation in the method code associated with a method name can arise through overriding during inheritance, or, given object interface types, different objects matching the type may have entirely independent and unrelated implementations.

Now let us discuss in more detail various OO language features from the perspective of how well they might fit into the functional framework of a language like SML.

In general, discussions of OO language design are based either on a ‘pure’ OO perspective, where OO concepts are the core and basis of the design (e.g. Smalltalk, Java), or they are viewed as extensions to a conventional, first-order imperative language such as Algol, C, Pascal or Modula. When we look at OO features from the perspective of a functional language like ML, we ask different questions and may make different judgments. In particular, we start with simple, powerful higher-order mechanisms, and so the addition of objects as another form of higher-order value is not such a major breakthrough. Let us consider which OO language features seem to offer something lacking in functional programming, and then we can consider the cost of adopting such features.

3.1. Encapsulation

Note that a functional language like ML has its own form of encapsulation in the form of function closures, which can be seen as ‘encapsulating’ their closure environment. Collections of functions defined in the same

[4] We ignore the fact that some OO languages allow direct access to state variables of objects, since this is generally considered a bad policy.

environment share closure environments, so one can view such a collection as an interface for accessing the closure environment. Scoping techniques (let or local declarations) can ensure that elements of the shared environment are not accessible except through the functions, as in the following example:

```
fun mkCounter n =
    let val r = ref n
     in get = fn () => !r,
        inc = fn () => r := !r + 1
    end
```

3.2. Dynamic Dispatch

All functional languages have dynamic dispatch, because this is just a matter of calling a computed function value (e.g. a function parameter of a higher-order function). The code executed when a function value is applied is not statically evident, but depends on the dynamic value of the function. Thus dynamic dispatch is really an inherent property of any higher-order language.

3.3. Objects

The simplest form of object is just a record of functions that share a common closure environment that carries the object state (we can call these *simple objects*). The function members of the record may or may not be defined as mutually recursive. However, if one wants to support inheritance with overriding, the structure of objects becomes more complicated. To enable open recursion, the call-graph of the method functions cannot be hard-wired, but needs to be implemented indirectly, via object self-reference. Object self-reference can be achieved either by construction, making each object a recursive, self-referential value (the *fixed-point model*), or dynamically, by passing the object as an extra argument on each method call (the *self-application* or *self-passing* model).[5] In either case, we will call these *self-referential objects*.

3.4. Classes

Classes have two uses: (1) as templates that are instantiated to create objects, and (2) as the base for deriving new subclasses using inheritance. The creation of simple objects is easy to implement in SML by writing functions that generate object values. The creation of self-referential objects in SML is problematical, requiring either a generalised version of `let rec` to create recursive object values, as in:

```
let val r = ref 0
    val rec self = {get = fn () => !r, inc = fn () => r := self.get() + 1}
 in #inc(self)(); #get(self)() end
```

or a datatype wrapped around the object type for self-passing, as in:

```
datatype counter = C of {get: C -> unit -> int, inc: C -> unit -> unit}
let val r = ref 0
    fun sendget (c as C{get,...}) = get c
    fun sendinc (c as C{inc,...}) = inc c
    val c = C{get = fn _ => fn () => !r, inc = fn Cget,... => r := get() + 1}
 in sendinc c (); sendget c () end
```

Another question about classes is whether a class is a value (in which case classes can be passed as parameters, stored in references, etc). Also, if a class is a value, what is its type? Do we need to enrich the type system with new type constructs to express class types? If classes are not values, are they modules? If not, classes are an entirely new kind of language entity whose dynamic and static relations with values, types and modules has to be specified.

[5] See [KaR94] for denotational descriptions of these two models. The fixed-point model first appeared in [Car84], while the self-application model was introduced in [Kam88].

Classes and inheritance may be modelled in various ways in SML, as will be discussed in Section 5, but with considerable syntactic overhead and with the loss of the automatic nature of code sharing associated with inheritance in OO languages.

3.5. Typing Issues

Should object types be pure interface types, with structural equivalence and subtyping, or should they be nominal class types? Since structural subtyping may not always agree with inheritance, it seems preferable to use interface types. In developing typed models of OO programming, it has generally been the practice to work with interface types, meaning types that specify the interface but not the implementation of an object (i.e. the type does not determine the class of an object). On the other hand, in most real-world OO languages, going back to Simula 67, classes have been treated as types. Recently languages like Java have supported both class types and interface types.

One peculiarity of class types is that they seem to tie the type to a particular implementation, but because of inheritance an object of a subclass may actually share no implementation code at all with a superclass type of which it is a member. Thus, unlike abstract types, class types generally do not carry any behavioural guarantees.

Although structural typing is more compatible in style with the ML type system, adding structural subtyping to the ML type system in an unrestricted way would be a major change, significantly complicating type inference and tending to produce type schemes encumbered with potentially large sets of type inequality constraints [AiW93, EST95, TrS96, OSW96, FaF97, Pot96, Pot98]. Some way of restricting either subtyping or type inference is needed to preserve the simplicity of Hindley–Milner type inference. One could use explicit coercion functions (as in [ThT94], or require coercion annotations as in OCaml, or restrict subtyping to a modified version of datatypes as in the OML proposal [ReR96b].

A second typing issue associated with classes is how much flexibility is allowed for changing method types with inheritance. Can method types be specialised in subclasses? Is a *selftype* identifier provided to refer implicitly to the current object type? Method specialisation requires a more complex underlying type system; bounded polymorphism can handle positive selftype occurrences, while binary methods need F-bounded polymorphism [CCH89].

3.6. Modularity Issues

In some OO languages, such as Java, classes are overloaded to serve as modules as well as object creation templates. In a language with a separate module system, this is unnecessary and undesirable.

Class systems also tend to come with more or less elaborate access control features, such as public/private/protected keywords. ML is equipped with scoping and information-hiding mechanisms like let and local and module signature matching that should minimise the need for special-purpose access control mechanisms for classes. In fact, a central principle of the Moby language design [FiR99] is to eliminate such redundancies between a class system and module system.

If classes were first-class values, then the relation between classes and modules would be clear, since classes would be value components and their signature specifications would be ordinary value specs. If classes are a new category of construct, then naturally they will also be a new category of module component with a corresponding signature specification. Functor abstraction will apply to modules containing classes, leading to functors with formal class parameters. In the body of such functors, the formal class parameter will be used as usual for object creation and subclass derivation, and the formal class specification must support static checking of such uses.

3.7. Open Recursion

The power of inheritance is based on the fact that it provides a form of open recursion. But open recursion raises serious methodological problems. The methods of a class are implicitly mutually recursive. When a method is overridden (i.e. replaced by a new definition in a subclass), not only may that method's behaviour change, but the behaviour of any other method that directly or indirectly calls the overridden method may change. Here is a simple example of this phenomenon:[6]

```
O1 = {x = fn () => 1, y = fn () => (2*self.x())}
O2 = {x = fn () => 1, y = fn () => 2}

  O1.x() ⟹ 1    O1.y() ⟹ 2
  O2.x() ⟹ 1    O2.y() ⟹ 2

O1' = O1 with {x = fn () => 2}
O2' = O2 with {x = fn () => 2}

  O1'.x() ⟹ 2    O1'.y() ⟹ 4
  O2'.x() ⟹ 2    O2'.y() ⟹ 2
```

Here `O1` and `O2` are two objects with the same interface and the same behaviour, but different internal ‘wiring’. If we apply the same method override to `O1` and `O2`, replacing the method `x`, we get different behaviours in the resulting `O1'` and `O2'`. Note that the non-overridden `y` method changes its behaviour, but only in `O1'`.

The point of this example is that, in order to understand the effect of a method override, we need to be aware of the internal call graph among methods. With inheritance and *super*, to determine this call graph we may need to examine the entire class hierarchy chain from its root to the class in question. Alternatively, one could give complete and rigorous behavioural specifications for each method (e.g. in terms of preconditions, postconditions, invariants and modifies lists) and prove that every method satisfies its specification based on the specifications of the methods it calls. Then if any overriding method is required to satisfy the same specification as the method it overrides, we could guarantee that the override cannot cause any method (including itself) to violate its specification. But this is a very tight restriction on overriding; presumably, overriding is often intended to change the behaviour of objects in significant ways.

This problem has led many OO authorities to deprecate inheritance or downgrade its importance. For instance, Gamma et al. ([GHJ95], page 20) state as their second principle of object-oriented design, ‘*Favor object composition over class inheritance*’, and Cardelli [Car96] explains the drawbacks of inheritance. But at the same time inheritance is the main technique for adaptability and reuse in OO programs. It plays a role similar to parameterisation, in that any method is a potential parameter to be instantiated by overriding. In theory, not having to specify in advance what attributes will change gives one more flexibility than explicit, prespecified parameters, but the trade-off is that it is more difficult to anticipate and control the effect of modifications. In summary, to quote Bob Harper, classes are a modularity construct that turns out to be highly *nonmodular*!

4. Models of Object-Oriented Programming

The problem of giving a precise, formal meaning to the naive metaphor underlying object-oriented programming and then embodying the formal analysis in a well-founded, statically typed language has occupied language researchers for much of the past two decades. This task has been surprisingly difficult, and has led to a number of alternative models of various degrees of complexity and supporting varying sets of language features. One general conclusion we can draw from the results of this modelling effort is that there is no obvious or simple way to capture the object-oriented metaphor in a principled language design; the underlying concepts and mechanisms are intrinsically difficult. We can try to illustrate this by looking at a fragment of one of the simpler and more intuitively accessible models proposed, the recursive record model originally developed by Kamin, Reddy and Cook [Kam88, Red88, KaR94, CoP95, CCH89].

Most models of typed OO programming can be expressed in the common framework of $F^{\omega}_{\leq}$ enriched with records, value-level and type-level recursion, references and other constructs. Bruce, Cardelli and Pierce [BCP99] provide a useful survey covering several models, and Fisher and Mitchell [FiM96] have written a tutorial covering the first two of these plus an axiomatic approach, and discussing their limitations. The first four chapters of Abadi and Cardelli [AbC96] give a general overview of language features and programming issues for object-oriented programming.

[6] This example uses a mixture of SML notation for functions, conventional OO dot notation for method invocation, and a record concatenation operation `with`.

The models covered in [BCP99] are characterised by the structure used for object types, with the simplest of these being a recursive record type.[7] We can express these object types in the form `Rec(X)I(X)`, where I(X) expresses the method interface as a record type potentially containing recursive references to the object type.

The associated encoding of classes is illustrated by the following class (expressed in a simple pseudo-code) that defines a cell containing an integer value `x` and two methods `get` and `equal`.

```
class C1(n: Int)
  var x: Int = n
  method getx() = x
  method equal(c: SelfType) = self.getx() = c.getx()
end
```

The interface type function `CF1` and corresponding object type `C1` given by

```
CF1 = Fun(X){getx: Unit -> Int, equal: X -> Bool}
C1 = Rec(X)CF1(X)
```

This is a functional class, meaning that its state (the instance variable `x`), is immutable. The object state representation is the record type:

```
CR1 = {x: Int}
```

The class `C1` itself is modelled by the following generator function, `c1gen`.

```
c1gen: All(X <: CF1(X))(CR1 -> X -> CF1(X)) =
  fun(X <: CF1(X))
    fun(r: CR1)
      fun(self: X)
        {getx = fun()r.x,
         equal = fun(c: X)(self.getx() = c.getx())}
```

Objects of class C1 are created by instantiating `c1gen` at `C1`, applying the result to the initial object state of type `CR1` and taking the fixed point of the resulting generator function.

```
newc1 : Int -> C1 = fun(n: Int)(fix(c1gen[C1](x=n)))
c1obj = newc1 3
```

The polymorphic type parameter `X` plays the role of '`SelfType`'. Note that `c1gen` is F-bounded polymorphic with bound `CF1`; this is to support inheritance in the presence of binary methods like `equal`. Even if `C1` did not have a binary method itself, F-bounded polymorphism would still be necessary to support addition of binary methods in subclasses.

Inheritance is modelled by defining a new class generator function in terms of c1gen. Consider the subclass `C2` defined by adding a new field `y` and a corresponding access method `gety`, and overriding the `equal` method of `C1`.

```
class C2(n: Int, m: Int)
  inherit C1(n)
  var y: Int = m
  method gety() = y
  method equal(c: SelfType) = super(c) & self.gety() = c.gety()
end
```

The derived interface function, object type, representation type and generator function for `C2` are defined as follows (using the record concatenation operators + for types and `with` for values):

```
type CF2 = Fun(X)(CF1(X) + {gety: Unit -> Int})
type C2 = Rec(X)CF2(X)
type CR2 = CR1 + {y: Int}
```

[7] We will follow [BCP99] in using a Cardelli-style synactic sugaring of $F^{\omega}_{<}$.

```
c2gen: All(X <: CF2(X))(CR2 -> X -> CF2(X)) =
  fun(X <: CF2(X))
    fun(r: CR2)
      fun(self: X)
        let super: CF1(X) = c1gen[X](r)(self)  (* using CR2 <: CR1 *)
         in super with
            {gety = fun()r.y,
             equal = fun(c: X)(super.equal(c) & self.gety() = c.gety())}
```

Note the definition of the `super` method record in terms of the `c1gen` class function. Object creation with `c2gen` follows the same standard pattern as used with `c1gen`:

```
newc1 : Int -> Int -> C2 =  fun(n:Int)fun(m:Int)(fix(c2gen[C2](x=n,y=m)))
c2obj = newc2 3 4
```

If a class is used recursively to create new objects within its methods this model needs to be further complicated by wrapping the object generator function inside a class generator function, and then taking an extra fixed-point to achieve the class recursion.

This example is fairly simple, though it does exercise the advanced feature of method specialisation for binary (contravariant) methods (like the `equal` method here), which is not found in conventional OO languages. But note the sophisticated machinery that was required. We need three levels of recursion: (1) in the construction of objects; (2) in the construction of object types; and (3) in the construction of classes that call themselves. We needed F-bounded polymorphism to allow for contravariant method specialisation in future subclasses.

The recursive record model is probably the most elementary and accessible of the object models. Let us summarise what OO features we can explain in this model:

- objects;
- object (interface) types;
- classes;
- new;
- inheritance;
- super;
- covariant method specialisation;
- contravariant method specialisation (binary methods);
- SelfType.

There are many more features found in popular OO languages that are not accounted for, and which require even more complicated types and encodings.

The recursive record model is arguably the most direct $F^{\omega}_{\leq}$-based model for OO programming, and it accounts for only a limited number of features. Yet the encoding techniques used for the recursive record model are clearly quite complex. Other models require coding techniques of similar or greater complexity, though which features can be captured and the difficulty of modelling individual features vary from one model to another.

The question then arises as to whether there are simpler and more direct models of OO languages that are not based on typed lambda calculi. Abadi and Cardelli address this question in [AbC96], where they develop a series of ‘object calculi’ with primitive constructs meant to more directly correspond to the elements of OO languages. It turns out that their main calculus has an equivalent typed lambda calculus formulation [ACV96], so it is doubtful that their object calculus is inherently simpler. The more direct mapping from OO languages to the object calculi may be balanced by intrinsic complexities in the object calculi themselves.

So the simple sugared language used for class definitions above hides a surprisingly subtle and complex set of representations and mechanisms. The question is whether programmers can somehow develop an accurate and reliable ‘working model’ of how classes and objects work without having to face these complexities.

5. Experiments in Adding OO and ML

Since there is some common ground between typed functional languages like ML and object-oriented languages, and since features of both styles of language can be modelled in a common framework like

$F^{\omega}_{<}$ (which is itself a rich functional language), the question naturally arises as to whether we can combine functional and object-oriented programming support effectively and gracefully in a single language. One could approach this issue from three different directions: adding functional features to an OO language, adding OO features to a functional language, or designing a new language from scratch with a primary goal of integrating both models. Here we are interested in the question of how one might add OO features to SML, and whether the result would be an improvement.

First we note that there ways to program in an object-oriented style in SML as it is. At the value level, we can represent objects as records of functions, as in this simple counter example.

```
type counter = {get : unit -> int, inc: unit -> unit}
fun mkCounter (n): counter =
    let val r = ref n
     in {get = fn () => !r,
         inc = fn () => r := !r + 1}
    end
 val mkCounter : int -> {get : unit -> int, inc: unit -> unit}
```

Here the object state is represented by the binding of `r` in the shared closure environment of the `get` and `inc` 'method' functions, and this makes it difficult to share or reuse the code of these methods. We could factor out the object state and define *pre-methods* [AbC96] that could potentially be used in other objects:

```
fun get_pre (r : int ref) = fn () => !r
fun inc_pre (r : int ref) = fn () => r := !r + 1
fun mkCounter (n): counter =
    let val r = ref n
     in {get = get_pre r,
          inc = inc_pre r}
    end
```

To create recursive object types we need to wrap the object record in a datatype. Odersky [Ode91] suggested a similar technique for encapsulating the state component of objects in a functional object scheme. Applied to the counter example, this would result in

```
datatype counter = Counter of {get : unit -> int, inc: unit -> counter}
type state = int ref
fun counter (g: state -> unit -> int, i: state -> unit -> state) s =
    Counter({get=g s, inc=(counter(g,i) o i) s})
fun get_pre (r : state) () = !r
fun inc_pre (r : state) () = r := !r + 1
val c = counter(get_pre,inc_pre)
```

Odersky's encoding idea was used extensively in the implementation of the eXene GUI library for SML/NJ [GaR93].

If we continue on in this vein, we might arrive at a collection of OO coding techniques similar to those proposed by Thorup and Tofte [ThT94]. They show how to emulate a significant set of OO features in SML without any language extensions. They use datatypes wrapped around records of functions to represent objects and explicit coercions to emulate subtyping.

One of their chief innovations is to approximate F-bounded polymorphism using ML-style polymorphism in conjunction with a *wrapper* type, which implements the interface specified by the type function F in terms of an underlying object type (*wrapping*), and makes it possible to retrieve (*unwrap*) the original object as well. The wrap function requires explicit code that implements the F interface in terms of the original object.

Thorup and Tofte also show how to emulate classes and inheritance using SML structures to represent classes. The superclass is nested as a substructure of the subclass structure, and the superclass state record is nested as a component of the subclass state (if there are new instance variables). Wrapper types are used to support inheritance by defining analogues of F-bounded polymorphic pre-methods. All inherited methods must be explicitly (re)defined, and this redefinition may involve coercing the state record and wrapping and unwrapping references to self. Object self-reference is implemented using recursive functions that produce a new copy of the self object at every self-reference, while sharing the object state.

The heaviness of the explicit encoding machinery rather negates the 'magic' of OO programming, particularly inheritance. Some simple language extensions (such as support for general value recursion to implement self-reference) and syntactic sugar might make this encoding more acceptable, but this attempt is more a demonstration of what can be done in principle than a practical basis for object-oriented programming in SML.

5.1. Object ML

Reppy and Riecke proposed a set of extensions in two papers [ReR96a, ReR96b] on Object ML (OML for short). OML adds subtyping based on a declared hierarchy of object types. Object types are similar to single constructor datatypes, except that new object types can be declared as extensions of previously defined object types.

```
objtype a = A of {| x : int |}
objtype b is a = B of {| y : int |}
```

The argument of an object constructor is a special kind of record including fields and methods, and an extension 'inherits' the fields of its parent, so the `b` object type has two fields named `x` and `y`.

Pattern matching is extended to object types and provides a simple form of *typecase* for dispatching off the type of an object, as in the following example:

```
fun f(B{| x, y |}) = x + y
  | f(A{| x |}) = x
 val f : a -> int
```

Pattern matching on object constructors can also be viewed as a kind of checked downcasting local to the function clause in question.

Object type declarations in a program construct an explicit subtyping hierarchy in which each object type has a finite linear chain of supertypes starting with its immediate parent object type. This makes checking the subtype relation between object types simple and efficient. Object types are nominal, and subtyping is based on declared relationships.

Objects can have *methods* as well as fields as members. Methods are declared as in the following example, which also illustrates that object types can be recursive:

```
objtype point = PT of {| x : int, eq: point -> bool |}
fun mkPoint x0 =
  PT {| x = x0, eq = meth self p => ($x self = PT$x pt) |}
val mkPoint : int -> point
```

Method invocation uses self-passing semantics, so if `p, q: point`, then `PT$eq p q` selects the `eq` method of `p` and implicitly passes `p` as the self parameter before passing `q`. Object types specify interface and not implementation, and objects can be defined directly by expressions using the object constructor.

A special `selftype` variable can be used in object type declarations to represent the type of the 'current' object, and its meaning varies between an object type and its derivatives. Selftype is restricted to appear in positive positions.

As in Thorup and Tofte's encoding, classes are modelled as structures containing pre-methods and object constructor functions. Inheritance involves reusing the pre-methods of the parent structure, which is supported by introducing a restricted form of bounded polymorphism (in [ReR96a] where bounds must be object types. Access control (hiding members of an object) can be achieved using *partially abstract* signature specifications for object types of the form '`objtype b is a`'.

Although bounded polymorphism makes it easier and more natural to share code between a class structure and a subclass, it is still necessary to explicitly define each method of the subclass, including inherited methods. There is no implicit inheritance of method code nor of field initialisation code, which can be considered a major drawback for OO programming.

However, even if the object and class features of OML are not judged to provide a satisfactory basis for object-oriented programming, the idea of hierarchical, extensible datatypes suggested by OML object types is an attractive option for enriching SML. As suggested in [ReR96b], they would have a number of uses, such as hierarchical classification of exceptions, and open-ended heterogeneous collections.

5.2. Objective Caml

The Objective Caml (OCaml) dialect of ML [Ler96] incorporates a class system based on a record model but using Rémy's 'row' polymorphism [Rem94b] as a partial substitute for true subtyping, combined with equirecursive type expressions [Pie02], Chapter 21). A structural form of subtyping is also supported, but through explicit coercions rather than the subsumption rule.

OCaml's object types are special record types that specify only the types of methods; fields are always private to an object. Recursive object types are expressed using an equirecursive type construct:

```
type point = <getx: int; move: int -> 'a> as 'a
```

Polymorphism over object types can be expressed using *open* object types containing an anonymous row variable denoted by '..':

```
fun (p: <getx: int; move: int -> 'a; .. > as 'a) -> p#getx;;
```

This function can be applied to points as well as other objects that contain additional methods, even if the type of the extended object is not a structural subtype of point. In this sense, row polymorphism is similar in effect to F-bounded polymorphism. These equirecursive, open object types can be inferred by the type checker.

Standalone object values can be defined, but objects are usually created by instantiating a class. Class definitions use a fairly conventional syntax, except that they can be wrapped with let declarations and can take parameters. Classes are not first-class values, but there is a notion of class types which can be used to constrain a class interface or specify a class in a module signature. Classes can be polymorphic, and the type parameter can be 'bounded' by requiring it to be an instance of an open object type (similar to an F-bound). A class declaration also implicitly declares an object type that is inferred from the class body, and an open (row polymorphic) version of the object type. Inferred object types can get quite large, so the size of printed types is kept manageable by a discipline of preserving type abbreviations wherever possible during type checking. OCaml manages to preserve Hindley–Milner style type inference and principal typings while adding row polymorphism and equirecursion.

The OCaml object/class system is very feature-rich. Here is a partial enumeration of the features supported:

- recursive object types;
- open, row-polymorphic object types;
- standalone object creation;
- classes;
- parameterised classes (class functions);
- class types;
- polymorphic class types (with type parameter constraints);
- self variable in method definitions (programmer selected name);
- self type variable (programmer selected name);
- multiple inheritance;
- multiple super variables (programmer selected names);
- virtual classes and methods;
- private methods;
- binary methods;
- hiding of inherited fields in subclass;
- single constructor per class, using class name;
- anonymous initialisation 'methods';
- object cloning with modification;
- explicit coercion from structural subtypes to supertypes.

Method specialisation and polymorphic methods are not (currently) supported.[8]

[8] Polymorphic methods are planned for a future release.

The class system is reasonably orthogonal to the module system. The module system can be used to control access through class type specification. Classes can be components of modules, and therefore can appear as formal parameters in functors within which they can be instantiated and subclassed.

In many respects, the OCaml class system seems to be a very successful extension of an ML base language with a very full and featureful class-based object system. The new features are conservative, in that they do not invalidate any old programs that do not use the object system, and OCaml manages to preserve the convenience of Hindley–Milner type inference while significantly enhancing the expressive power of the type system.

There are some problems, however. The size and complexity of the type system of the full language are formidable, and although simplified and abstracted fragements have been described in detail [ReV98], the full type system does not seem to have been rigorously defined. Acquiring a full and accurate understanding of the complete type system is also likely to be a major challenge for the average OCaml programmer, and the result will be that most programmers will develop only a partial and empirical understanding of the type system and type checking.

There are some stylistic clashes introduced by the extensions. For instance, a separate imperative-style sublanguage has been introduced just for use within class bodies. More generally, the object system supports and encourages an imperative style of programming which is at odds with the 'value-oriented' style that has evolved in the ML community and which is one of the major advantages of the language. Informal sampling of the OCaml user community indicates the most OCaml programmers stick to the traditional functional core and use the object system only in special circumstances, if at all.

Finally, the use of row-polymorphism plus equirecursion is a very clever alternative to the type checking problems posed by true subsumption, but the basic techniques underlying type checking of these features are rather subtle and non-obvious to the average user. For instance, type checking often requires one to relate a closed recursive object type to an open recursive object type constraint, as in the followiing example:

```
let f(x: <get: int; id: 'a; .. > as 'a) = x#id#get;;
let g(y: <get: int; id: 'a; eq: 'a -> bool; .. > as 'a) = f(x#id#get);;
```

The implicit row variable represented by '`..`' is implicitly bound at the outermost level in the type of `f`:

```
f : (All 'r) <get: int; id: 'a; .. > as 'a -> int
```

Naively, one might think that when `f` is applied to `y` in the body of g, the row type variable `'r` gets instantiated to `< eq: 'a -> bool >`, but this would involve a free variable capture. So what is really happening is that `'r` is instantiated to `< eq: t -> bool >`, where `t` is the closed object type of `y`. Thus typechecking must verify the following system of equations:

```
t = <get: int; id: 'a; eq: 'a -> bool > as 'a -> int
s = <get: int; id: 'a; eq: t -> bool > as 'a -> int
s = t
```

The coinductive proofs of such equations are not difficult for type theorists, but they are beyond what we typically expect working programmers to master.

To summarise, the arguments against OCaml as a satisfactory solution to unifying ML and object-oriented programming are that the language and its type system are too large and complex to be mastered by typical users, the feature set is too heavily weighted on the object-oriented side in comparison with how the whole language is best used, and the suble interactions between row polymorphism and equirecursion will escape most programmers, who will fall back on a plausible but inaccurate working model.

5.3. Multimethod-Based Schemes for ML

Duggan [Dug94], Millstein and Chambers [MiC02], and Bonniot [Bon02] have all proposed object-oriented extensions of ML based on the idea of multimethods, which are method-like functions that can dispatch on the types of more than one argument. For instance, in the Millstein–Chambers version, one is allowed to extend previously defined datatypes with new constructors and then extend functions over the previous datatype with new clauses to handle the new constructors. This approach is reminiscent of Burstall's language NPL [Bur77], which was a precursor of Hope that provided open-ended datatypes and functions.

The main drawback of such schemes is that the addition of new constructors and function clauses involves side effects on previously defined functions, even if they were defined in separate modules. The semantics of such an extension is complex, and it is a major departure from the traditional lambda calculus-derived semantics of ML. It would also require major changes in the way functions are implemented.

5.4. $F^{\omega}_{\leq}$-Based Language Designs

Since much research on modelling OO features is based on extended versions of $F^{\omega}_{\leq}$, it is natural to consider an extended $F^{\omega}_{\leq}$ as an alternative to ML as a basis for unifying functional and object-oriented programming. The core of such a language features impredicative (first-class) polymorphism, higher-order type operators and quantifiers and subtyping, and this core is typically extended with records, value and type recursion, and state. Cardelli's Quest [CaL91] and Fun [Car91] languages are essentially syntactically sugared versions of $F^{\omega}_{\leq}$, much in the way that Landin's ISWIM was a sugared version of the lambda calculus. In such languages, as the modeling research shows, one can explicitly encode many OO constructs and features, and one could easily add additional syntactic sugar supporting OO programming.

More recently, Fisher and Reppy have been developing a language called Moby that has a full object class system integrated into a functional framework with first-class polymorphism, subtyping and ML-like modules. The object model is based on Fisher's 'protocols' [Fis96], another variant of objects as records. A principal design goal of Moby is to separate the roles of classes and modules and to eliminate redundant features. The class system focuses narrowly on the role of a class as an object template (object view) and base for deriving subclasses (class view). Managing access and visibility is done through a combination of declaration keywords (public vs. private fields and methods) and scoping mechanisms of the SML-like module system.

In Moby, object types are similar to recursive record types and contain specifications for visible fields and methods. Subtyping and type equality for object types are structural (though [FiR00] proposes to add nominal class types as well). Moby has no downcasting or type case mechanism for dynamically dispatching on the class of an object, but hierarchical, extensible datatypes (inspired by the object types of OML) can be used to tag objects and then programs can dispatch on these tags.

As in OCaml, class definitions determine associated object types and class interface types can be defined that provide the necessary information for both object views and class views. Moby provides several of the conventional keywords for controlling visibility and inheritance of fields and methods: *public*, *final* and *abstract* (similar to virtual).

Class definitions use the special variable `self` for object self-reference and `super` for reference to superclass members, but unlike OCaml there is no selftype, and inheritance does not support method type specialisation. On the other hand, in Moby, class interfaces can be used to hide even public class members, and then the restricted class can be used as the base for further inheritance.

Even with the separation of modularity mechanisms from classes, the class system of Moby is not very lightweight. It still delivers a feature set almost as large as OCaml's. It differs from OCaml mainly in using structural subtyping with subsumption and a predicative polymorphic type system (first-class polymorphism). Thus the core language of Moby resembles a subset of $F^{\omega}_{\leq}$ more than ML. This means that Hindley–Milner type reconstruction must be replaced by something like local type inference [PiT98, HoP99, OZZ01].

5.5. Summary of Extension Experiments

While simple objects are easily implemented in SML, going beyond these to emulate a full class system with inheritance leads to heavy encoding machinery and the sharing of implementation code; that is, the main point of inheritance is not automatic but requires explicit code to reuse and adapt methods of a superclass. Even with moderate extensions of the type system such as those provided in Object ML, the coding of classes as modules is generally felt to be too cumbersome.

So other designs like OCaml and Moby start by adding classes as a new basic construct, complete with a long list of conventional features borrowed from class-based object-oriented languages. The resulting type systems are large and the features added to support conventional object-oriented programming interact in subtle and non-obvious ways. In fact, at this point neither OCaml nor Moby has a full, formal description of its type system.

6. Conclusions

Contrasting the elements of a functional programming language like ML with even a clean and idealised object-oriented language, we find that the functional language generally provides simpler, lighter-weight structures: functions vs. objects, records and datatypes vs. objects, parametric polymorphism vs. subtyping, and lambda abstracton vs. inheritance. We also have enough machinery in an impure language like ML to mimic the more basic OO constructs and programming techniques, but with some encoding overhead.

The strengths of the paradigms are somewhat complementary, so it seems worthwhile to try to unify the models. Clearly, adding subtyping to functional programming would be a useful addition (as in $F^{\omega}_{\leq}$), though it complicates type inference. Perhaps one can substitute row-polymorphism (as used in OCaml) and get most of the benefit. Adding classes with inheritance to FP is less compelling because of the methodological problems associated with open recursion and the inherent complexities exposed by the various formal models.

A general question we should ask about an object system in a functional language like ML is how extensively it would be used. Much theoretical research on object systems has been concerned with ‘functional’ objects for the sake of semantic simplicity, and functional object systems actually stress the type system more than imperative ones, because more methods return objects. But the overwhelming preponderance of actual object-oriented programming involves imperative programming with stateful objects. It is reasonable to assume that the value-oriented style would still be preferred in most circumstances, and that objects, being typically stateful, would be used in more or less the same limited circumstances as imperative constructs are used currently. If this is the case, any object-oriented extensions should not drastically distort the language.

Of course, one could speculate on how usage of the language might change if an object-oriented extension were added – in a mixed paradigm language the dominant paradigm might shift over time – but the experience so far with OCaml seems to indicate that this would not happen with Standard ML.

Finally, there is a language complexity budget to be considered. Simple languages are good. The experiments so far seem to indicate that adding a full-blown object/class system to a language like ML could more than double the size and complexity of the type system, which is too large a burden to place on programmers.

Perhaps the best approach would be to add a few modest extensions to a functional language that would make it more convenient to program lightweight approximations to object-oriented structures and algorithms. A restricted form of record subtyping and hierarchical datatypes as in OML would be the leading candidates for such extensions, and such an extension was proposed in [MWG99]. Row-polymorphic record types, as in OCaml, but without equirecursion, would be another reasonable alternative.

References

[AbC96] Abadi, M. and Cardelli, L.: *A Theory of Objects*. Springer, 1996.

[ACV96] Abadi, M., Cardelli, L. and Viswanathan, R.: An interpretation of objects and object types. In *POPL ’96: 23rd ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 396–409, January 1996.

[AiW93] Aiken, A. and Wimmers, E. L.: Type inclusion constraints and type inference. In *ACM Symposium on Functional Programming Languages and Computer Architecture (FPCA)*, pages 31–41, 1993.

[App95] Appel, A.: Value-oriented programming. PowerPoint presentation, 1995.

[BCP99] Bruce, K. B., Cardelli, L. and Pierce, B. C.: Comparing object encodings. *Information and Computation*, 155(1/2):108–133, 1999.

[BFP97] Bruce, K. B., Fiech, A. and Petersen, L.: Subtyping is not a good ‘match’ for object-oriented languages. In *ECOOP ’97*, volume 1241 of *LNCS*, pages 104–127. Springer, 1997.

[BuG77] Burstall, R. and Goguen, J.: Putting theories together to make specifications. In R. Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058, 1977.

[BuG80] Burstall, R. and Goguen, J.: The semantics of Clear, a specification language. In *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, volume 86 of *Lecture Notes in Computer Science*, pages 292–332. Springer, 1980.

[Bon02] Bonniot, D.: Type-checking multi-methods in ml (a modular approach). In *9th Workshop on Foundations of Object-Oriented Languages*, pages 14–27, 2002.

[Bur69] Burstall, R.: Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[Bur77] Burstall, R.: Design considerations for a functional programming language. In *Infotech State of the Art Conference, The Software Revolution*, Copenhagen, 1977.

[BMS80] Burstall, R. M., MacQueen, D. B. and Sannella, D.: Hope: an experimental applicative language. In *Conference Record of the 1980 Lisp Conference*, pages 136–143, August 1980.

[Car84] Cardelli, L.: A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *LNCS*, pages 104–127. Springer, 1984.

[Car89] Cardelli, L.: Typeful programming. Technical Report 45, Digital Systems Research Centre, May 1989.

[Car91] Cardelli, L.: Typeful programming. In E. J. Neuhold and M. Paul, editors, *Formal Description of Programming Concepts*. Springer, 1991.

[Car96] Cardelli, L.: Bad engineering properties of object-oriented languages. *Computing Surveys*, 28(4es), 1996.

[Car97] Cardelli, L.: Type systems. In A. B. Tucker, editor, *Handbook of Computer Science and Engineering*, pages 2208–2236. CRC Press, 1997.

[CaL91] Cardelli, L. and Longo, G.: A semantic basis for quest. *Journal of Functional Programming*, 1(4):417–458, 1991.

[CaW85] Cardelli, L. and Wegner, P.: On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1985.

[CCH89] Channing, P., Cook, W., Hill, W., Orloff, W. and Mitchell, J.: F-bounded quantification for object-oriented programming. In *Proceedings Fourth International Conference on Functional Programming and Computer Architecture*, pages 273–280, September 1989.

[CoP95] Cook, W. and Palsberg, J.: A denotational semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1995.

[Dug94] Duggan, D.: Object interfaces, polymorphic methods and multi-method dispatch for ml-like languages. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 50–61, June 1994.

[EST95] Eifrig, J., Smith, S. and Trifonov, V.: Type inference for recursively constrained types and its application to oop. In *Proceedings of the 1995 Mathematical Foundations of Programming Semantics Conference*, volume 1 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.

[FaF97] Flanagan, C. and Felleisen, M.: Componential set-based analysis. *ACM SIGPLAN Notices*, 32(5):235–248, 1997.

[Fis96] Fisher, K.: *Type Systems for Object-oriented Programming*. PhD thesis, Stanford University, 1996.

[FiM96] Fisher, K. and Mitchell, J. C.: The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1(3):189–220, 1996.

[FiR99] Fisher, K. and Reppy, J.: The design of a class mechanism for moby. In *Proceedings of the 1999 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1–13, May 1999.

[FiR00] Fisher, K. and Reppy, J.: Inheritance-based subtyping. In *Seventh Workshop on Foundations of Object-Oriented Languages*, January 2000.

[GHJ95] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. M.: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[GaR93] Gansner, E. R. and Reppy, J. H.: A multi-threaded higher-order user interface toolkit. In *Software Trends*, volume 1, pages 61–80. Wiley, 1993.

[HaR99] Hansen,M. R. and Rischell, H.: *Introduction to Programming in SML*. Addison-Wesley, 1999.

[Har] Harper, R.: Introduction to Standard ML. http://foxnet.cs.cmu.edu/intro-notes.ps.

[HoP95] Hofmann, M. and Pierce, B.: A unifying type-theoretic framework for objects. *Journal of Functional Programming*, 5(4):593–635, 1995.

[HoP99] Hosoya, H. and Pierce, B. C.: How good is local type inference? Technical Report MS-CIS-99-17, University of Pennsylvania, June 1999.

[Hug89] Hughes, J.: Why functional programming matters. *Computer Journal*, 32(2):98–107, 1989.

[Kam88] Kamin, S.: Inheritance in smalltalk 80: a denotational definition. In *ACM Symposium on Principles of Programming Languages (POPL)*, pages 80–87, 1988.

[KaR94] Kamin, S. N. and Reddy, U. S.: Two semantic models of object oriented languages. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*, pages 464–495. MIT Press, 1994.

[Lan66] Landin, P.: The next 700 programming languages. *Communications of the ACM*, 9(3):157–166, 1966.

[Ler96] Leroy, X.: The objective caml system. Software and documentation available at `http://www.ocaml.org`, 1996.

[Mac81] MacQueen, D. B.: Structure and parameterisation in a typed functional language. In *Proceedings of the 1981 Symposium on Functional Languages and Computer Architecture*, pages 524–538, June 1981. Gothenburg, Sweden.

[MiC02] Millstein, T. and Chambers, C.: Modular typechecking for hierarchically extensible datatypes and functions. In *9th Workshop on Foundations of Object-Oriented Languages*, pages 1–13, 2002.

[Mil78] Milner, R.: A theory of type polymorphism in programming. *Journal of Computer and System Scieces*, 17:348–375, 1978.

[MWG99] ML2000 Working Group. Principles and a preliminary design for ML2000. http://research.bell-labs.com/dbm/papers/ml2000.ps, March 1999.

[Ode91] Odersky, M.: Objects and subtyping in a functional perspective. Technical Report RC 16423, IBM Research, 1991.

[OSW96] Odersky, M., Sulzmann, M. and Wehr, M.: Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1):35–55, 1996.

[OZZ01] Odersky, M., Zenger, C. and Zenger, M.: Colored local type inference. In *Proceedings of 28th ACM Symposium on Principles of Programming Languages*, pages 41–53, January 2001.

[Pau96] Paulson, L. C.: *ML for the Working Programmer*, 2nd edition. Cambridge University Press, 1996.

[Pie02] Pierce, B. C.: *Types and Programming Languages*. MIT Press, 2002.

[PiT94] Pierce, B. C. and Turner, D. N.: Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, 1994.

[PiT98] Pierce, B. C. and Turner, D. N.: Local type inference. In *Proceedings of 25th ACM Symposium on Principles of Programming Languages*, pages 252–265, January 1998.

[Pot96] Pottier, F.: Simplifying subtype constraints. In *International Conference on Functional Programming (ICFP)*, pages 122–133, 1996.

[Pot98] Pottier, F.: A framework for type inference with subtyping. In *International Conference on Functional Programming (ICFP)*, pages 228–238, 1998.

[Red88] Reddy, U.: Objects as closures: abstract semantics of object-oriented langauges. In *Proceedings of the 1988 ACM Conference on Lisp and Functional Programming*, pages 289–297, July 1988.

[Rem94a] Rémy, D.: Programming objects with ML-ART: An extension to ML with abstract and record types. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *LNCS*, pages 321–346. Springer, 1994.

[Rem94b] Rémy, D.: Type inference for records in a natural extension of ML. In C. A. Gunter and J. C. Mitchell, editors, *Theoretical Aspects of Object-Oriented Programming: Types, Semantics, and Language Design*. MIT Press, 1994.

[ReV98] Rémy, D. and Vouillon, J.: Objective ml: An effective object-oriented extension to ml. *Theory and Practice of Object Systems*, 4(1):27–50, 1998.

[ReR96a] Reppy, J. H. and Riecke, J. G.: Classes in object ml via modules. In *Proceedings of the Third Workshop on Foundations of Object-Oriented Languages*, July 1996.

[ReR96b] Reppy, J. H. and Riecke, J. G.: Simple objects for standard ml. In *Proceedings of the 1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 171–180, May 1996.

[Rey74] Reynolds, J. C.: Towards a theory of type structure. In *Proceedings Colloque sur la Programmation*, volume 19 of *LNCS*, pages 408–425. Springer, 1974.

[Tof96] Tofte, M.: Essentials of Standard ML Modules. 1996 Summer School on Advanced Functional Programming, Oregon Graduate Institute, August 1996.

[TrS96] Trifonov, V. and Smith, S.: Subtyping constrained types. In *Proceedings of the Third International Static Analysis Symposium*, volume 1145 of *LNCS*, pages 349–365. Springer, 1996.

[ThT94] Thorup, L. and Tofte, M.: Object oriented programming and Standard ML. In *Record of the 1994 ACM SIGPLAN Workshop on ML and its Applications*, pages 41–49, June 1994.

Received March 2002
Accepted April 2002