On Understanding Data Abstraction, Revisited

William R. Cook

University of Texas at Austin

wcook@cs.utexas.edu

Abstract

In 1985 Luca Cardelli and Peter Wegner, my advisor, published an ACM Computing Surveys paper called "On understanding types, data abstraction, and polymorphism". Their work kicked off a flood of research on semantics and type theory for object-oriented programming, which continues to this day. Despite 25 years of research, there is still widespread confusion about the two forms of data abstraction, *abstract data types* and *objects*. This essay attempts to explain the differences and also why the differences matter.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features—Abstract data types; D.3.3 [*Programming Languages*]: Language Constructs and Features—Classes and objects

General Terms Languages

Keywords object, class, abstract data type, ADT

1. Introduction

What is the relationship between *objects* and *abstract data types* (ADTs)? I have asked this question to many groups of computer scientists over the last 20 years. I usually ask it at dinner, or over drinks. The typical response is a variant of "objects are a kind of abstract data type".

This response is consistent with most programming language textbooks. Tucker and Noonan [57] write "A class is itself an abstract data type". Pratt and Zelkowitz [51] intermix discussion of Ada, C++, Java, and Smalltalk as if they were all slight variations on the same idea. Sebesta [54] writes "the abstract data types in object-oriented languages... are called classes." He uses "abstract data types" and "data abstraction" as synonyms. Scott [53] describes objects in detail, but does not mention abstract data types other than giving a reasonable discussion of opaque types.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.

So what is the point of asking this question? Everyone knows the answer. It's in the textbooks. The answer may be a little fuzzy, but nobody feels that it's a big issue. If I didn't press the issue, everyone would nod and the conversation would move on to more important topics. But I do press the issue. I don't say it, but they can tell I have an agenda.

My point is that the textbooks mentioned above are wrong! Objects and abstract data types are not the same thing, and neither one is a variation of the other. They are fundamentally different and in many ways complementary, in that the strengths of one are the weaknesses of the other. The issues are obscured by the fact that most modern programming languages support both objects and abstract data types, often blending them together into one syntactic form. But syntactic blending does not erase fundamental semantic differences which affect flexibility, extensibility, safety and performance of programs. Therefore, to use modern programming languages effectively, one should understand the fundamental difference between objects and abstract data types.

While objects and ADTs are fundamentally different, they are both forms of *data abstraction*. The general concept of data abstraction refers to any mechanism for hiding the implementation details of data. The concept of data abstraction has existed long before the term "data abstraction" came into existence. In mathematics, there is a long history of abstract representations for data. As a simple example, consider the representation of integer sets. Two standard approaches to describe sets abstractly are as an *algebra* or as a *characteristic function*. An algebra has a sort, or collection of abstract values, and operations to manipulate the values[1]. The characteristic function for a set maps a domain of values to a boolean value, which indicates whether or not the value is included in the set. These two traditions in mathematics correspond closely to the two forms of data abstraction in programming: algebras relate to abstract data types, while characteristic functions are a form of object.

In the rest of this essay, I elaborate on this example to explain the differences between objects and ADTs. The

[1] The sort, or carrier set, of an algebra is often described as a set, making this definition circular. Our goal is to define specific set abstractions with restricted operations, which may be based on and assume a more general concept of sets

examples focus on non-mutable objects, because they are sufficient to explain the main points. Other topics, including inheritance and reflection, are also ignored in order to focus on the basic idea of data abstraction.

When I'm inciting discussion of this topic over drinks, I don't tell the the full story up front. It is more fun to keep asking questions as the group explores the topic. It is a lively discussion, because most of these ideas are documented in the literature and all the basic facts are known. What is interesting is that the conclusions to be drawn from the facts are not as widely known. Most groups eventually work through the differences between objects and ADTs, but I can tell they walk away feeling uneasy, as if some familiar signposts now point in different directions. One source of unease is that the fundamental distinctions are obscured, but not eliminated, in real programming languages. Also, the story is quite complex and multi-faceted. This essay is only an introduction to a large body of literature on the relationship between objects and ADTs.

In my conversations about objects and ADTs, my next step is to push the discussion towards a more precise understanding of data abstraction. What is an abstract data type? What is an object? For abstract data types, there is general agreement.

2. Abstract Data Types

An *abstract data type* (ADT) has a public name, a hidden representation, and operations to create, combine, and observe values of the abstraction. The familiar built-in types in most languages, for example the `int` and `bool` data types in Algol, Pascal, ML, Java and Haskell, are abstract data types.

In addition to built-in abstract data types, some languages support user-defined abstract data types. User-defined abstract data types that resemble built-in data types were first realized in CLU [37, 36] and Alphard [61] in the 1970s. There were also strong connections to algebraic specification of data types [24, 7] The core ideas introduced in CLU were adapted for ML [42], Ada [49], Modula-2 [60]. As an example, Figure 1 defines an abstraction for integer sets, adapted from the CLU reference manual [36].

```
set = cluster is empty, contains, insert
rep = oneof[empty: null,
            pair: struct[first:int, rest:rep]]
empty = proc() returns (cvt)
   return(rep$make_empty(nil));
  end empty;
insert = proc(s: cvt, i: int) returns (cvt)
   if  contains(up(s), i) then
     return(rep$make_pair(pair$(first:i,rest:s))
   else
     return(s);
   end
  end insert
isEmpty = proc(s: cvt) returns (bool)
   typecase s
     tag empty: return(true)
     tag pair(p:pair): return(false);
   end
  end empty;
contains = proc(s: cvt, i: int)
   typecase s
     tag empty: return(false)
     tag pair(p:pair):
       if p.first = i then return(true)
       else return(contains(up(p.rest), i))
     end
   end contains
union = proc(s1: cvt, s2: cvt)
   typecase s1
     tag empty: return(s2)
     tag pair(p:pair):
       return insert(union(up(p.rest), s2), p.first)
     end
   end union
  end set
```

Figure 1. CLU cluster for integer sets

The representation type is a list of integers. In discussions of CLU, these values are called "objects" or "data objects", although they are not necessarily the same as objects in object-oriented programming.

CLU used explicit syntax and operators to manage the hiding of the representation. The `cvt` type represents the public view of the representation type, while the functions `up` and `down` convert between public and private views of the type. Rather than explain CLU in detail, it is easier to give the same abstraction in ML, as in Figure 2, where the hiding mechanism is simplified and type inference simplifies the types.

Figure 3 gives the *signature* of the resulting abstract data type. A signature defines the type name (but not its representation) and the types of the operations. The signature can be extended with a full specification of the behavior of integer sets. Abstract data types support very powerful specification and verification techniques, including equational theories [20, 3, 7] and axiomatic specifications [26, 40, 17]. The specifications work well in this context; they are intuitive, elegant and sound.

Clients can declare values of type set and use operations to manipulate the values.

```
let a = empty()
    b = insert(a, 3)
in
    if contains(b, 2) then "yes" else "no"
```

```
abstype set = EMPTY | INS of int * set where
val empty = EMPTY
fun insert(s, i) =
  if not contains(s, i)
  then INS(i, s)
  else s
fun isEmpty(s) = (s == EMPTY)
fun contains(s, i) =
  case s of
    EMPTY ⇒ false
  | INS(n, r) ⇒
      if i = n then true
      else contains(r, i)
  end
fun union(s1, s2) =
  case s1 of
    EMPTY ⇒ s2
  | INS(n1, r1) ⇒ insert(union(r1, s2), n1)
  end
end
```

hide what's inside set:
now only interface to set
is isEmpty, contains, union

Figure 2. ML abstract data type (ADT) for integer sets

```
type set
val empty    : set
val isEmpty  : set → bool
val insert   : set × int → set
val contains : set × int → bool
val union    : set × set → set
```

Figure 3. Signature for integer set abstract data type

But clients cannot inspect the representation. This is why the isEmpty function is needed, because the following program is illegal when written outside of the abstraction:

```
fun test(a : set) = (a == EMPTY);
```

The function test is attempting to break the encapsulation of the data abstraction to peek at its internal representation. There is also no predefined notion of equality on integer sets. If equality is desired, it must be programmed and made explicit in the ADT interface.

2.1 Representation Independence

The name set is *abstract* because it has a public name but its details are hidden. This is a fundamental characteristic of abstraction: something is visible on the surface, but the details are hidden. In the case of *type abstraction*, the type name is public, but the representation is hidden. With procedural abstraction, the procedure interface (name and arguments) is public, but the operational details are hidden. Type abstraction is a technical mechanism that can be used to support data abstraction.

One of the practical benefits of data abstraction is that it allows internal implementation details to be changed without affecting the users of the abstraction. For example, we could modify the code for set to represent integer sets as hash tables or balanced binary trees. For example, Figure 4 is an alternative implementation based on a sorted list representation.

2.2 Optimization

A different implementation opens up the possibility for optimizing some of the operations. For example, the union operation in Figure 2 is quite expensive to compute. With a sorted list representation union is computed in linear time. Insertion is faster in some cases, but it may require copying more nodes. Deciding what representations to use, based on the associated algorithmic trade-offs, is a standard software engineering activity.

These optimizations depend critically upon an important feature of abstract data types: the ability to inspect the representation of more than one abstract value at the same time. Multiple representations are inspected in the union operation. There is nothing surprising about inspecting multiple representations. It is a natural side-effect of the type system and the fact that all values of type set belong to the abstract data type implementation that created them. As we shall see, the ability to inspect multiple representations does have some important consequences.

2.3 Unique Implementations

With ML abstypes, CLU clusters, Ada packages and Modula-2 modules there can only be one implementation of an abstract data type in any given program. The implementation is a construct that manages a collection of *values* that inhabit the type. All the values from a given implementation share the same representation type, although there can be multiple different representational variants within the type. This is usually accomplished by defining the representation type as a labeled sum. The type name set is a globally bound name that refers to a single hidden representation. The type system ensures that it is sound for the implementation to inspect any set value.

Having only one implementation of a data abstraction is limiting. There is already a name clash between the definitions in Figures 2 and 4. One of them had to be given a different name, set2, even though they are really just two different versions of the same abstraction. Client programs have to be edited to choose one or the other implementation.

ADTs are also frequently used in C programming [32], using header files as a simple module system. The signature of the type is given in a header file as a *forward reference* to a structure that is only defined in the implementation file. An example header file for integer sets is given in Figure 5. This trick works because the C compiler does not need to know

```
abstype set2 = EMPTY | INS of int * set2 where

val empty = EMPTY

fun insert(s, i) =
  case s of
    EMPTY ⇒ INS(i, s)
  | INS(n, r) ⇒
      if i = n then s
      else if i < n then INS(i, s)
      else let t = insert(r, i) in
        if r = t then s else INS(n, t)

fun isEmpty(s) = (s == EMPTY)

fun contains(s, i) =
  case s of
    EMPTY ⇒ false
  | INS(n, r) ⇒
      if i = n then true
      else if i > n then false
      else contains(r, i)
  end

fun union(s1, s2) =
  case s1 of
    EMPTY ⇒ s2
  | INS(n1, r1) ⇒
      case s2 of
        EMPTY ⇒ s1
      | INS(n2, r2) ⇒
          if n1 = n2 then
            insert(n1, union(r1, r2))
          else if n1 < n2 then
            insert(n1, union(r1, s2))
          else
            insert(n2, union(s1, r2))
      end
    end
end
```

Figure 4. Integer set ADT with sorted list representation

```
struct set_rep; // representation is not defined in header
typedef struct set_rep* set;
set   empty();
bool  isEmpty(set s);
set   insert(set s, int i);
bool  contains(set s, int i);
set   union(set s1, set s2);
```

Figure 5. Abstract data type in C header file

```
type SetImp = ∃ rep . {
  empty    : rep,
  isEmpty  : rep → bool,
  insert   : rep × Int → rep,
  contains : rep × Int → Bool,
  union    : rep × rep → rep
}
```

Figure 6. Type of first-class ADT set implementations

the format of the representation type, it only needs to know the size of a pointer to the representation.

2.4 Module Systems

The problem of unique implementation is solved by putting abstract data types into modules. ML [39] has a module system that allows multiple implementations for a given signature. The signature of an abstraction can be defined once, and multiple implementations written in separate modules. A client program can then be parameterized over the signature, so that a particular implementation can be selected during module binding. There can be multiple implementations in software repository, but one implementation is used in a given program.

Allowing multiple implementations is good, but it is still not as flexible as might be desired. Consider a case where one part of a program needs to use the sorted list representation for integer sets, and another part of the program needs to use a binary tree representation. Having two different implementations for an abstraction is possible in ML, Ada, or Module-2. However, the two different parts of the program cannot interoperate. The different parts of the program cannot *exchange* integer sets. As a result the following program is illegal:

```
fun f(a : set, b : set2) = union(a, b)
```

There is no union operation to combine a set with a set2. Given the signature we have defined, it is not even possible to write such an operation.

The ML module system also allows multiple inter-related abstract types to be defined in a single module. For example, a personnel application might have data abstractions *Employee* and *Department* with operations to associate employees with departments.

2.5 Formal Models

Formal models of abstract data types are based on *existential types* [44]. In this model, ADT implementations are first class values with existential type, as defined in Figure 6.

A value of type SetImp is not a set, it is an implementation of a set abstraction. This two-level structure is essential to abstract data types: the first level is an implementation (SetImp) which publishes an abstract type name and a set of operations. Within that implementation, at the second level,

are the values that represent elements of the named abstract type (set).

This existential type is nearly identical to the signature in Figure 3. Intuitively, it asserts that "a type locally identified as rep exists such that the following operations are defined...".

Most practical languages do not support the full generality of first-class ADT implementations. Thus existential values and their usage are not familiar to most programmers. Explaining the mechanics of existential types is beyond the scope of this essay. They are described in Cardelli and Wegner's paper [10], and also covered thoroughly in Pierce's book, *Types and Programming Languages* [50].

To use an existential value, it must be *opened* to declare a name for the representation type and access the operations. Each time an existential value is opened, it creates a completely new type name. Thus if an ADT implementation is opened twice, the values from one instance cannot be mixed with values from the other instance. In practice, it is standard to open all ADTs once in the global scope of the program. The ML module system has more sophisticated sharing mechanisms that allow multiple implementations to coexist, while allowing interoperability between multiple uses of the same abstractions. Even in this case values from the two different implementations cannot be mixed.

2.6 Summary

An abstract data type is a structure that implements a new type by hiding the representation of the type and supplying operations to manipulate its values. There are several ways in which abstract data types seem fundamentally right.

- They work just like built-in types.
- They have sound proof techniques.
- ADTs can be implemented efficiently, even for complex operations that require inspection of multiple abstract values.
- From a type theory viewpoint, abstract data types have a fundamental model based on existential types. Existential types are the dual of universal types, which are the basis for parametric polymorphism (called generics in Java and C#). The duality of universal and existential types is fundamental, and it leaves little room for any other alternative. What else could there be?
- There is a solid connection to mathematics. An ADT has the same form as an abstract algebra: a type name representing an abstract set of values together with operations on the values. The operations can be unary, binary, multi-ary, or nullary (that is, constructors) and they are all treated uniformly.

All of these observations lead to the general conclusion that abstract data types are *the* way to define data abstractions. This belief is so deep-seated, so obviously correct, that it is almost impossible to think of any alternative. Many people take "abstract data type" and "data abstraction" as synonyms.

But abstract data types are not the only way to define data abstractions. The alternative is fundamentally different.

3. Objects

Object-oriented programming has its origin in the language Simula 67 [16]. Zilles published a paper describing a form of objects [62] before he started working with Liskov and switched his focus to ADTs. At the same time, Smalltalk [55, 28], Actors [25] and Scheme [56, 1] all explored objects in an untyped setting. Smalltalk especially formulated these ideas into a philosophically and practically compelling language and environment for object-oriented programming. As these languages were all dynamically typed, they did not immediately contribute to the ongoing dialog about statically typed data abstraction in the form of ADTs.

There is not a single universally accepted model of object-oriented programming. The model that I present here is recognized as valid by experts in the field, although there certainly are other valid models. In particular, I present objects in a denotational style which I believe exposes their core concepts in an intuitive way. I believe that operational approaches obscure the essential insights.

In this section I discuss a pure form of object-oriented programming with interfaces [9, 8]. The practical realities of popular languages are discussed in Section 5.

To begin with, let us reconsider the idea of integer sets. One alternative way to formulate integer sets is as the characteristic function:

type ISet = Int → Boolean

The type Int → Boolean is the type of functions from integer to boolean. It is clear that this is a different way to think about sets than the abstract data types presented in the previous section. Consider a few values of this type:

Empty = λi. false
Insert(s, n) = λi. (i = n **or** s(i))
Union(s_1, s_2) = λi. (s_1(i) **or** s_2(i))

The expression $\lambda i.e$ represents a function with a parameter named i and a result expression e. The empty set is just a function that always returns false. Inserting n into a set s creates a function that tests for equality with n or membership in the functional set s. Given these definitions, it is easy to create and manipulate sets:

a = Insert(Empty, 1)
b = Insert(a, 3)
print a(3) – *results in true*

In what sense could ISet be understood as defining a data abstraction for integer sets? We have been conditioned to think in terms of representations and operations. But these

```
interface ISet = {
  isEmpty   : bool,
  contains  : int → bool,
  insert    : int → ISet,
  union     : ISet → ISet
}
```

Figure 7. Object-oriented integer set interface

concepts do not apply in this case. One might say that this approach represents sets as functions from integers to booleans. But this 'representation' looks like an interface, not a concrete representation.

Note that there is no "contains" operation, because the set itself *is* the contains operation. Although it may not seem like it, the characteristic function is the pure object-oriented approach to defining integer sets. You may not accept this statement immediately, because I have not talked about any classes, methods, or inheritance, which are supposed to be characteristic of objects.

3.1 Object Interfaces

ISet is an object-oriented interface to an integer set data abstraction. The function is an observation of the set, and a set is 'represented' by the observations that can be performed upon it. One problem with this interface is that there is no way to tell if the set is empty. A more complete interface is given in Figure 7. It is a record type with four components corresponding to methods. The field names of the record are capitalized, to distinguish them from other uses of the same names. The result is a standard object-oriented interface for immutable integer set objects.

An essential observation is that *object interfaces do not use type abstraction*: there is no *type* whose name is known but representation is hidden. The type ISet is defined as a record type containing functions from known types to known types. Instead, objects use procedural abstraction to hide behavior. This difference has significant consequences for use of the two forms of data abstraction.

Object interfaces are essentially higher-order types, in the same sense that passing functions as values is higher-order. Any time an object is passed as a value, or returned as a value, the object-oriented program is passing functions as values and returning functions as values. The fact that the functions are collected into records and called methods is irrelevant. As a result, the typical object-oriented program makes far more use of higher-order values than many functional programs.

The empty operation in the ADT is not part of the object-oriented ISet interface. This is because it is not an observation on sets, it is a constructor of sets.

u is like "let rec" for values

```
Empty = μ this. {
  isEmpty   = true,
  contains  = λi. false
  insert    = λi. Insert(this, i)
  union     = λs. s
}

Insert(s, n) = if s(n) then s else μ this. {
  isEmpty   = false,
  contains  = λi. (i = n or s(i))
  insert    = λi. Insert(this, i)
  union     = λs. Union(this, s)
}

Union(s1, s2) = μ this. {
  isEmpty   = false,
  contains  = λi. (s1(i) or s2(i))
  insert    = λi. Insert(this, i)
  union     = λs. Union(this, s)
}
```

Figure 8. Object-oriented integer set implementations

3.2 Classes

Several implementations for the ISet interface are defined in Figure 8. The contains method is the same as the simple functions given above. The definitions have the same types, after redefining ISet.

The special symbol μ is used to define recursive values [50]. The syntax $\mu x.f$ defines a recursive value where the name x can appear in the expression f. The meaning of $\mu x.f$ is the value of f where occurences of x represent recursive references within f to itself. Objects are almost always self-referential values, so every object definition uses μ. As a convention, we use this as the name x, but any name could be used. The bound name x corresponds to self in Smalltalk or this in C++.

Each of these definitions correspond to a *class* in object-oriented programming. In this encoding, classes are only used to construct objects. The use of classes as types is discussed later.

The definition of class state, or member variables, is different from Java [21]. In this encoding, the member variables are listed as parameters on the class, as in Scala [47].

Several of the method bodies are repeated in these definitions. The insert method simply invokes the Insert class to create a new ISet object with one more member. Inheritance could be used to reuse a single method definition. Inheritance is often mentioned as one of the essential characteristics of object-oriented programming. However, inheritance will not be used in this section because it is neither necessary for, nor specific to, object-oriented programming [13].

A client of these classes looks just like a Java program, with the familiar method invocation style:

```
Empty.insert(3).union(Empty.insert(1))
  .insert(5).contains(4)
```

Selecting a function to invoke from a record containing function values is usually called *dynamic binding*. This term is not a very intuitive description of what is essentially an invocation of a higher-order function.

Just as the ADT version of integer sets had two levels (set implementations and set values), the object-oriented version has two levels as well: interfaces and classes. A class is a procedure that returns a value satisfying an interface. Although Java allows class constructors to be overloaded with more than one definition, it is clear that one of the primary purposes of a class is to construct objects.

3.3 Autognosis

A careful examination of the union operator in the object interface, in Figure 7, reveals that the parameter is typed by an interface. This means that the union method in a set object cannot know the representation of the other set being unioned. Fortunately, the union operator does not need to know the representation of other sets, it just needs to be able to test membership. The Union class in Figure 8 constructs an object that represents the union of two sets s_1 and s_2.

To me, the prohibition of inspecting the representation of other objects is one of the defining characteristics of object-oriented programming. I term this the *autognostic* principle:

> *An object can only access other objects through their public interfaces.*

Autognosis means 'self knowledge'. An autognostic object can only have detailed knowledge of itself. All other objects are abstract.

The converse is quite useful: any programming model that allows inspection of the representation of more than one abstraction at a time is not object-oriented.

One of the most pure object-oriented programming models yet defined is the Component Object Model (COM) [5, 22]. It enforces all of these principles rigorously. Programming in COM is very flexible and powerful as a result. There is no built-in notion of equality. There is no way to determine if an object is an instance of a given class.

Autognosis has a profound impact on the software engineering properties of a system. In particular, an autognostic system is much more flexible. But at the same time, it can be more difficult to optimize operations. More significantly, there can be subtle relationships between the public interface of a class and the ability to implement behavior, as discussed in Section 3.5.

3.4 Flexibility

Object interfaces do not prescribe a specific representation for values, but instead accept any value that implements the required methods. As a result, objects are flexible and extensible with new representations. The flexibility of object interfaces can be illustrated easily by defining several new kinds of set. For example, the set of all even integers, and the set of all integers, are easily defined:

```
Even = μ this. {
  isEmpty   = false,
  contains  = λi. (i mod 2 = 0)
  insert    = λi. Insert(this, i)
  union     = λs. Union(this, s)
}

Full = μ this. {
  isEmpty   = false,
  contains  = λi. true
  insert    = λi. this
  union     = λs. this
}
```

The Full set returns itself as the result of any insert or union operation. This example also illustrates that objects can easily represent infinite sets easily.

These new sets can be intermixed with the sets defined above. Other specialized sets can also be defined, including the set of prime numbers or sets representing intervals.

```
Interval(n, m) = μ this. {
  isEmpty   = (n > m),
  contains  = λi. (n ≤ i and i ≤ m)
  insert    = λi. Insert(this, i)
  union     = λs. Union(this, s)
}
```

There is no direct equivalent to this kind of flexibility when using abstract data types. This difference is fundamental: abstract data types have a private, protected representation type that prohibits tampering or extension. Objects have behavioral interfaces which allow definition of new implementations at any time.

The extensibility of objects does not depend upon inheritance, but rather is an inherent property of object interfaces.

3.5 Interface Trade-Offs

The choice of interfaces to an object can affect which operations are efficient, which are slow, and also which operations are impossible to define.

For example, it is not possible to augment the integer set interface with an intersect operation, because it is not possible to determine if the intersection of two sets is empty without iterating over the sets. It is commonplace to include iterator methods in collection classes like the ones given here. But iterators do not interact well with infinite sets. Significant software engineering decisions must be made when designing interfaces, but these issues are rarely discussed in programming language textbooks.

One problem with object interfaces is that efficiency considerations often allow implementation issues to influence the design of interfaces. Adding public methods that inspect the hidden representation can significantly improve efficiency. But it also restricts the flexibility and extensibility of the resulting interface.

3.6 Optimization

The optimization of the union method based on sorted lists is not possible in the object-oriented implementation, without modifying the interfaces. The optimization would be possible if the interfaces included a method to iterate the set contents in sorted order. Extending an object interface with more public methods can significantly improve performance, but it also tends to reduce flexibility. If the sets used a more sophisticated representation, optimizations might require more representational details to be exposed in the public interface.

There are several optimizations in the object implementation in Figure 8. The first is that the union method on empty sets is the identity function. The second is that the insert class does not always construct a new value. It only creates a new value if the number being inserted is not in the set already.

It is not necessary to include insert and union as methods inside the object interface, because they can be defined as classes that operate on any sets. The optimization of union in the empty set class is one reason why it is useful to internalize the creation operations in the object interface.

3.7 Simulation

Object-oriented programming was first invented in the context of the simulation language Simula [16, 4]. The original intent was to simulate real-world systems, but I believe that simulation also allows one object to simulate, or pretend to be, another object.

For example, the set Interval(2, 5) simulates a set that has integers 2 through 5 inserted into it. According to the principle of autognosis, there should be no way for any part of the program to distinguish between the interval and the inserted set. There are many operations that violate this principle, including pointer equality and instanceof tests.

Simulation also provides a basis for verification of object-oriented programs. If two objects simulate each other, forming a *bisimulation*, then they are equivalent [41]. The concept of simulation and bisimulation are powerful mathematical concepts for analyzing the behaviors.

3.8 Specifications and Verification

Object-oriented programming has caused significant problems for verification efforts [34, 45, 2]. This is not surprising if you understand that object-oriented programming is high-order procedural programming; objects are a form of first-class procedure value, which are passed as arguments and returned as values everywhere. It is difficult to verify programs that combine first-class higher-order functions and imperative state.

A common complaint is that it is impossible to determine what code will execute when invoking a method. This is no different from common uses of first-class functions. If this objection is taken seriously, then similar complaints must be leveled against ML and Haskell, because it is impossible (in general) to determine what code will run when invoking a function value.

More significantly, it is possible to create bad objects easily. For example, the following object does not meet the specification for integer sets:

$$
\begin{array}{l}
\mathsf{bad} = \mu\,\mathsf{this}.\ \{ \\
\quad \mathsf{IsEmpty} \quad = (\mathsf{random}() > 0.5), \\
\quad \mathsf{Contains} \quad = \lambda \mathsf{i}.\ (\mathsf{time}()\ \mathbf{mod}\ \mathsf{i} = 1) \\
\quad \mathsf{Insert} \quad = \lambda \mathsf{i}.\ \mathsf{this} \\
\quad \mathsf{Union} \quad = \lambda \mathsf{s}.\ \mathsf{Insert}(3,\ \mathsf{s}) \\
\}
\end{array}
$$

It reports that it is empty 50% of the time, and includes integers randomly based on time of day. Object interfaces can be given behavioral specifications, which can be verified to prohibit bad objects.

A more subtle problem is that objects do not necessarily encapsulate state effectively [27]. The problem arises when the state of an object is itself a collection of objects. There is a tendency for the internal objects to leak out and become external, at which point the abstract boundary is lost. This problem motivates the ongoing research effort on ownership types [6].

One particularly difficult problem is that methods can be re-entered while they are running [46]. This causes problems for the standard Hoare-style approach to verification. In this approach, the class enforces an invariant, and every procedure (method) is given a precondition and a post-condition. The problem is that any method calls within the body of the method may loop back around and invoke some other method of the object being verified. In this case the other method may be called while the object is in an inconsistent state. It may also modify the object state, to invalidate the assumptions used to verify the original method.

Abstract data types do not usually have this problem because they are built in layers; each layer invokes lower layers, but lower layers do not invoke higher layers. Not all systems can be organized in this fashion, however. Complex systems often require *notifications*, or call-backs, which allow lower layers to call into higher layers. This can cause problems for verification if call-backs are included in ADTs.

Object-oriented programming is designed to be as flexible as possible. It is almost as if it were designed to be as difficult to verify as possible.

3.9 Some More Theory

The object interface has some interesting relationships to the abstract data type signature in Figures 3 and 6. First, the

methods have one fewer argument than the corresponding operations in the ADT signature. In each case, the rep argument is missing. Second, the rep in the ADT operations corresponds to a *recursive* reference to ISet in each method of the object interface. The similarity can be expressed by the following type function:

$$
\begin{array}{l}
\textbf{type } \mathsf{F}(t) = \{ \\
\quad \mathsf{IsEmpty} \quad : \mathsf{bool}, \\
\quad \mathsf{Contains} \ : \mathsf{int} \to \mathsf{bool}, \\
\quad \mathsf{Insert} \quad : \mathsf{int} \to t, \\
\quad \mathsf{Union} \quad : t \to t \\
\}
\end{array}
$$

The types given above can be rewritten in terms of F:

$$
\begin{array}{l}
\mathsf{ISet} = \mathsf{F}(\mathsf{ISet}) \\
\mathsf{SetImp} = \exists\ \mathsf{rep}.\ \mathsf{rep} \times (\mathsf{rep} \to \mathsf{F}(\mathsf{rep}))
\end{array}
$$

The original definition of SetImp is isomorphic to this new definition. To see the relationship, note that in $\mathsf{rep} \to \mathsf{F}(\mathsf{rep})$ the function type with domain rep supplies the missing argument that appears in all the ADT operations. The cartesian product with rep supplies the empty constructor.

The definition of SetImp above is the encoding of a final coalgebra $X \to F(X)$ into the polymorphic λ-calculus [19]. The only problem is that F is not a covariant functor, because of the union method. This encoding also corresponds to the greatest fixedpoint of F, which corresponds to the recursive type ISet. The relationship between coalgebra and objects is an active research topic [29].

3.10 Summary

An object is a value exporting a procedural interface to data or behavior. Objects use procedural abstraction for information hiding, not type abstraction. Object and and their types are often recursive. Objects provide a simple and powerful form of data abstraction. They can be understood as closures, first-class modules, records of functions, or processes. Objects can also be used for procedural abstraction.

Unlike abstract data types, many people find objects to be deeply disturbing. They are fundamentally higher-order, unlike abstract data types. With an object, you are never quite certain what it is going to do: What method is being called? What kind of object is it really?

On the other hand, many people find objects to be deeply appealing in their simplicity and flexibility. They do not require complex type systems. Inheritance allows recursive values to be extended in powerful ways.

The fact that objects are autognostic, so that they can only know themselves, is also confusing. On the one hand, it interferes with desirable optimizations that require inspection of multiple representations. One solution is to expose representational details in the object's interface, which limits flexibility. The benefits of autognosis are often subtle and only realized as a system grows and evolves.

Finally, as parts of a long and rich tradition of abstraction, objects too—not just ADTs—are fundamentally grounded in mathematics

4. Relationships between ADTs and OOP

Although object-oriented programming and abstract data types are two distinct forms of data abstraction, there are many relationships between them. Many simple abstractions can be implemented in either style, although the usages of the resulting programs is quite different.

4.1 Static Versus Dynamic Typing

One of the most significant differences between abstract data types and objects is that objects can be used to define data abstractions in a dynamically typed language.

Objects do not depend upon a static type system; all they need is some form of first-class functions or processes.

Abstract data types depend upon a static type system to enforce type abstraction. It is not an accident that dynamic languages use objects instead of user-defined abstract data types. Dynamic languages typically support built-in abstract data types for primitive types; the type abstraction here is enforced by the runtime system.

Type systems only enforce structural properties of programs; they do not ensure conformance to a specification. But with ADTs, the type system can ensure that if the ADT implementation is correct, then all programs based on it will operate correctly. The type system prevents outside clients from tampering with the implementation. Pure object interfaces allow any structurally compatible implementation, thus the type system does not prohibit bad implementations from being used.

4.2 Simple and Complex Operations

One point of overlap between objects and abstract data types is that simple data abstractions can be implemented equally well in either style. The difference between simple and complex data abstractions is whether or not they have operations, like the union operation in the set ADT, that inspect the representation of multiple abstract values.

In this essay I call an operation "complex" if it inspects multiple representations. In some of the literature complex operations are called "binary". Literally speaking, a binary operation is one that accepts two inputs of the abstract type. For an object, a binary method is one that takes a second value of the abstract type, in addition to the abstract value whose method is being invoked. According to these definitions, union is always binary.

However, not all binary methods are *complex*. This depends on how the operation is implemented. A binary operation can be implemented by invoking public methods on the abstract arguments. Doing so does not require the representation of the two values to be inspected. The union operation in Figures 1 and 2 are simple. But the union operation in Figure 4 is complex.

Pure object-oriented programming does not support complex operations. Doing so requires inspection of another object's representation, using instance-of or similar means.

Any abstract data type with only simple operations can be implemented without loss of functionality, but more simply and extensibly, with objects.

Consider an ADT implementation with the following type, where t does not appear in σ_i, τ_j, ρ_j, or δ_k.

$$\begin{array}{l}\mathsf{F(t)} = \{ \\ \qquad \mathsf{c}_i \ : \sigma_i \to \mathsf{t}, \\ \qquad \mathsf{o}_j \ : \mathsf{t} \times \tau_j \to \rho_j, \\ \qquad \mathsf{m}_k : \mathsf{t} \times \delta_k \to \mathsf{t} \\ \quad \} \\ \mathsf{ADT} : \exists\, \mathsf{t}.\mathsf{F(t)}\end{array}$$

The methods have been partitioned into *constructors*, *observations* and *mutators*. The constructors c_i create values of type t. The observations take an input of type t with additional arguments and produce values of some other type. The mutators take an input of type t and produce a result of type t. These patterns are exhaustive, because there are no complex methods. τ_j or δ_k is unit if there are no other arguments besides t for a given operation.

Create a new type I to represent the object interface:

$$\begin{array}{l}\mathsf{interface\ I} = \{ \\ \quad \mathsf{o}_j \ : \tau_j \to \rho_j, \\ \quad \mathsf{m}_k : \delta_k \to \mathsf{I} \\ \} \end{array}$$

For the constructors, define a family of functions that invoke a wrap function that creates the object. The notation for this example is that of Pierce's book *Types and Programming Languages* [50].

$$\begin{array}{l} C_i : \sigma_i \to T \\ C_i(\mathsf{x} : \sigma_i) = \\ \quad \textbf{let}\ \{*\mathsf{t},\ \mathsf{p}\} = \mathsf{ADT}\ \textbf{in} \\ \quad\quad \mathsf{wrap[t](p,\ p.c}_k\mathsf{(x))} \end{array}$$

$$\begin{array}{l} \mathsf{wrap} : \forall \mathsf{t}.\ \mathsf{F(t)} \to \mathsf{I} \\ \mathsf{wrap[t](p,\ x)} = \{ \\ \quad \mathsf{o}_j \ = \lambda \mathsf{a}{:}\tau_j.\ \mathsf{p.m}_j\mathsf{(x,\ a)}; \\ \quad \mathsf{m}_k = \lambda \mathsf{a}{:}\delta_k.\ \mathsf{wrap[t](p,\ p.m}_k\mathsf{(x,\ a))}; \\ \} \end{array}$$

The constructors first open the ADT, construct an appropriate value of type t and then wrap it as an object. This transformation is a direct corollary of the basic definitions of ADTs [44] and objects [13].

The converse, however, is not necessarily true. It is possible to take any fixed set of object-oriented classes that implement an interface and convert them to an ADT. One simple way to do it is to use objects as the representation type for the ADT, but rewriting the abstractions is always possible. However, the result is no longer extensible, so the conversion incurs a loss of flexibility.

4.3 Extensibility Problem

When implementing data abstractions, there are two important dimensions of extensibility. New representational variants can be added, or new operations can be added. This observation suggests it is natural to organize the behaviors into a matrix with representations on one axis and observations/actions on the other. Then extensibility can be viewed as adding a column or row to the matrix.

In the 1970s, as work began on understanding data abstraction, Reynolds published a prophetic paper that identified the key differences between objects and abstract data types [52, 23], although I think he did not realize he was describing objects. Reynolds noticed that abstract data types facilitate adding new operations, while "procedural data values" (objects) facilitate adding new representations. Since then, this duality has been independently discovered at least three times [18, 14, 33],

This duality has practical implications for programming [14]. Abstract data types define operations that collect together the behaviors for a given action. Objects organize the matrix the other way, collecting together all the actions associated with a given representation. It is easier to add new operations in an ADT, and new representations using objects. Although not discussed in detail here, object-oriented programs can use inheritance to add new operations [14].

Wadler later gave the problem a catchy name, the "Expression Problem", based on the well-known canonical example of a data abstraction for expressions with operations to print, evaluate, or perform other actions [58].

The extensibility problem has been solved in numerous ways, and it still inspires new work on extensibility of data abstractions [48, 15]. Multi-methods are another approach to this problem [11]. More complex variations, involving integration of independent extensions, have still not been completely resolved.

4.4 Imperative State and Polymorphism

Issues of imperative state and polymorphism have been avoided in this essay because they are, for the most part, orthogonal to the issues of data abstraction. The integer sets discussed in this paper can be generalized to polymorphic sets, set<t>. These generalization can be carried out for either abstract data types or objects. While there is significant work involved in doing so, the issues of polymorphism do not interact very much with the issues relating to data abstraction.

Both abstract data types and objects can be defined in either a pure functional or imperative style. Pure functional objects are quite common, although not as common as they could be. Issues of state are largely orthogonal from a language design viewpoint. However, imperative programming has a significant impact on verification.

5. Reality

The reality in practical programming languages is not so pure and simple. It turns out that statically typed object-oriented languages all support both pure objects and also a form of abstract data types. They also support various hybrids.

5.1 Object-Oriented Programming in Java

While Java is not a pure object-oriented language, it is possible to program in a pure object-oriented style by obeying the following rules

Classes only as constructors A class name may only be used after the keyword *new*.

No primitive equality The program must not use primitive equality (==). Primitive equality exposes representation and prevents simulation of one object by another.

In particular, classes may not be used as types to declare members, method arguments or return values. Only interfaces may be used as types. Also, classes may not be used in casts or to test with instanceof.

This is generally considered good object-oriented style. But what if you were *forced* to follow this style, because the language you were using required it? Smalltalk comes close. Since Smalltalk is dynamically typed, classes are only used as constructors. It does support instanceof, although it is rarely used.

One other way to break encapsulation in Java is through the use of *reflection*, although this is not common when writing most programs. Reflection is useful when writing meta-tools (e.g. debuggers) and program generators. However, use of reflection appears to be growing more widespread. More research is needed to quantify the effect of reflection on data abstraction and encapsulation.

5.2 ADTs in Java

It takes a little more work to encode abstract data types in statically typed object-oriented programming languages.

```
class ASet {
  // declare representation fields
  // no public constructor
  static ASet empty();
  static ASet insert(ASet s, int n);
  static bool  contains(ASet s, int n);
  static ASet union(ASet a, ASet b);
}
```

Using a *class* name as a type introduces type abstraction. A class hides its representation. Object-oriented languages do not always support the sums-of-products data structures found in other languages, but such types can be simulated using an abstract class with a subclass for each variant in the sum type. Pattern matching on these types can then be implemented by using instanceof and appropriate casts. One direct encoding uses static methods for all the ADT operations, and the class just holds the representation.

```
class CSet {
  // declare representation fields
  // no public constructor
  static CSet empty();
  CSet  insert(Integer n);
  bool  contains(Integer n);
  CSet  union(CSet b);
}
```

To summarize, when a class name is used as a type, it represents an abstract data type.

5.3 Haskell Type Classes

Type classes in Haskell [30] are a powerful mechanism for parameterization and extensibility [59]. A type class is an algebraic signature that associates a group of operations with one or more type names. A type class for integer sets, defined below, is very similar to the existential type in Figure 6, but in this case uses curried functions:

```
class Set s where
  empty   :: s
  isEmpty :: s → Bool
  insert  :: s → Int → s
  contains:: s → Int → Bool
  union   :: s → s → s
```

Functions can be written using the generic operations:

```
test :: Set s ⇒ s → Bool
test s = contains(union(insert(s, 3), insert(empty, 4)), 5)
```

The qualification on the type of test indicates that the type s is any instance of Set. Any type can made an *instance* of Set by defining the appropriate operations:

```
instance Set [Int] where
  empty    = []
  isEmpty  = (== [])
  insert   = flip (:)
  contains = flip elem
  union    = (++)
```

Instance definitions can connect type classes with actual types that come from different libraries, and all three parts can be written without prearranged knowledge of the others. As a result, type classes are flexible and extensible.

A type can only be an instance of a class in one way. For example, there is no way to define sorted lists and lists as both being different instances of Set. This restriction can always be bypassed by creating a new type that is a tagged or labeled version of an existing type, although this can introduce undesirable bookkeeping when tagging values.

Type classes are similar to object interfaces in allowing a method to operate on any value that has the necessary operations.

On the other hand, type classes are based on algebraic signatures as in abstract data types. The main difference is that type classes do not enforce any hiding of representations. As a result, they provide parametric abstraction over type signatures, without the information hiding aspect of ADTs. Given the success of Haskell, one might argue that encapsulation is somewhat overrated.

Type classes are not autognostic. When a function is qualified by a type class, the same type instance must be used for all values within that function. Type classes do not allow different instances to interoperate. There are other ways in which Haskell provides abstraction and information hiding, for example, by parametericity.

On the other hand, the object-oriented data abstractions given here can also be coded in Haskell. In addition, an existential type can be used to combine the type class operations with a value to create a form of object [31]. In this encoding, the type class acts as a method table for the value.

5.4 Smalltalk

There are many interesting aspects of the Smalltalk language and system. One curious fact is that Smalltalk has no built-in control flow and very few built in types. To see how this works, consider the Smalltalk implementation of Booleans.

There are two Boolean classes in Smalltalk, named True and False. They both implement a two-argument method called `ifTrue:ifFalse:`.

```
class True
   ifTrue: a ifFalse: b
        ^ a value

class False
   ifTrue: a ifFalse: b
        ^ b value
```

Method names in Smalltalk are sequences of keyword labels, where each keyword identifies a parameter. The body of the `True` method returns the result of sending the `value` message to the first argument, `a`. The body of the `False` method returns the second argument, `b`.

The `value` method is needed because `a` and `b` represent *thunks* or functions with a single dummy argument. A thunk is created by enclosing statements in square brackets. A conditional is implemented by sending two thunks to a Boolean value.

```
(x > y) ifTrue: [ x print ]
        ifFalse: [ y print ]
```

transfer object x to print object?

if this evals to true, True里的fun会被执行

The implementation of Booleans and conditionals in Smalltalk is exactly the same as for Church booleans in the λ-calculus [12]. Given that objects are the only way to implement data abstraction in an untyped language, it makes sense that the same kind of data would be used in Smalltalk and the untyped λ-calculus. It would be possible to implement a `RandomBoolean` class that acts as true or false based on the flip of a coin, or a `LoggingBoolean` that traced how many computations were performed. These booleans could be use anywhere that the standard booleans are used, including in low-level system code.

Smalltalk numbers are not Church numerals, although they share some characteristics. In particular, numbers in Smalltalk implement iteration, just as they do in the Church encoding. Similarly, Smalltalk collections implement a reduce operator analogous to the Church encoding of lists.

The Smalltalk system does include a primitive integer type, implemented as an ADT for efficiency. The primitive types are wrapped in high-level objects, which communicate with each to other through an ingenious interface to perform coercions and implement both fixed and infinite precision arithmetic. Even with these wrappers, I claim that Smalltalk is not truly "objects all the way down" because the implementation depends upon primitive ADTs. It may be that objects are simply not the best way to implement numbers. More analysis is needed to determine the efficiency costs and whether the resulting flexibility is useful in practice.

One conclusion you could draw from this analysis is that the untyped λ-calculus was the first object-oriented language.

6. Discussion

Academic computer science has generally not accepted the fact that there is another form of data abstraction besides abstract data types. Hence the textbooks give the classic stack ADT and then say "objects are another way to implement abstract data types". Sebesta focuses on imperative data abstractions without complex methods, using stacks as an example, so it is not surprising that he does not see any difference between objects and ADTs [54]. Tucker and Noonan also illustrate data abstraction with stacks [57]. But they also provide a Java implementation of a type-checker and evaluator that appears to have been translated directly from ML case statements, implemented using `intanceof` in Java. The resulting program is a poor illustration of the capabilities of object-oriented programming.

Some textbooks do better than others. Louden [38] and Mitchell [43] have the only books I found that describe the difference between objects and ADTs, although Mitchell does not go so far as to say that objects are a distinct kind of data abstraction.

The rise of objects interrupted a long-term project in academia to create a formal model of data based on ADTs. Several widely used languages were designed with ADTs as their fundamental form of data abstraction: ML, Ada, and Modula-2. As object-oriented programming became more prominent, these languages have adopted or experimented with objects.

Object-oriented programming has also been subject to extensive academic research. However, I believe the academic community as a whole has not adopted objects as warmly as

they were received in industry. I think there are three reasons for this situation. One is that the conceptual foundations for objects, discussed here, are not widely known. The second is that academics tend to be more interested in correctness than flexibility. Finally, programming language researchers tend to work with data abstractions that are more natural as ADTs.

There are significant design decisions involved in choosing whether to implement a given abstraction with ADTs or with objects. In her history of CLU [35], Barbara Liskov discussed many of these issues, and gave good arguments for her choice of the ADT style. For example, she writes that "although a program development support system must store many implementations of a type..., allowing multiple implementations within a single program seems less important." This may be true if the types in question are stacks and integer sets, but when the abstractions are windows, file systems, or device drivers, it is essential to allow multiple implementations running within the same system.

To me it is unfortunate that Liskov also wrote that "CLU is an object-oriented language in the sense that it focuses attention on the properties of data objects and encourages programs to be developed by considering abstract properties of data." I believe that there is no technically or historically meaningful sense in which CLU is an object-oriented language. I do believe that modern object-oriented languages have been influenced by CLU (especially in the encapsulation of representation) but this does not make CLU into an object-oriented language.

Acknowledgements

There are too many people to thank individually for all their discussions on the topic of this essay. I thank Olivier Danvy, Shriram Krishnamurthi, Doug Lea, Yannis Smaragdakis, Kasper Osterbye, and Gilad Bracha for their comments on the essay itself.

7. Conclusion

Objects and abstract data types (ADTs) are two different forms of data abstraction. They can both implement simple abstractions without complex methods, but objects are extensible while ADTs are easier to verify. Significant differences arise when implementing abstractions with complex operations, for example comparisons or composition operators. Object interfaces support the same level of flexibility, but often force a trade-off between interface simplicity and efficiency. Abstract data types support clean interfaces, optimization, and verification, but do not allow mixing or extending the abstractions. Mathematically oriented types, including numbers and sets, typically involve complex operations that manipulate multiple abstract values, and are best defined using ADTs. Most other types including files, device drivers, graphic objects, often do not require optimized complex operations, and so are best implemented as objects.

Modern object-oriented languages support a mixture of object-oriented and ADT functionality, allowing programmers to choose ADT style for specific situations. In modern object-oriented languages, the issue boils down to whether or not classes are used as types. In a pure object-oriented style, classes are only used to construct objects, and interfaces are used for types. When classes are used as types, the programmer is implicitly choosing to use a form of abstract data type. The decision affects how easy it is for the program to be extended and maintained over time, and also how easy it is to optimize complex operations. Understanding the fundamental differences between objects and ADTs can help in choosing to use them wisely.

References

[1] N. Adams and J. Rees. Object-oriented programming in Scheme. In *Proceedings of the ACM Conf. on Lisp and Functional Programming*, pages 277–288, 1988.

[2] P. America. A behavioral approach to subtyping object-oriented programming languages. In *Proceedings of the REX Workshop/School on the Foundations of Object-Oriented Languages*, volume 173 of *Lecture Notes in Computer Science*, 1990.

[3] J. Bergstra and J. Tucker. Initial and final algebra semantics for data type specifications: Two characterisation theorems. Research Report IW 142, Stichting Mathematisch Centrum, 1980.

[4] G. M. Birtwistle. *DEMOS: a system for discrete event modelling on Simula*. Springer-Verlag, 1987.

[5] D. Box. *Essential COM (DevelopMentor Series)*. Addison-Wesley Professional, 1998.

[6] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. *SIGPLAN Notices*, 38(1):213–223, 2003.

[7] R. Burstall and J. Goguen. Putting theories together to make specifications. In *International Joint Conferences on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[8] P. Canning, W. Cook, W. Hill, and W. Olthoff. Interfaces for strongly-typed object-oriented programming. In *Proceedings of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 457–467, 1989.

[9] L. Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*, pages 51–68. Springer-Verlag, 1984.

[10] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *Computing Surveys*, 17(4):471–522, 1986.

[11] C. Chambers. Object-oriented multi-methods in Cecil. In *ECOOP '92: Proceedings of the European Conference on Object-Oriented Programming*, pages 33–56. Springer-Verlag, 1992.

[12] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[13] W. Cook. *A Denotational Semantics of Inheritance*. PhD thesis, Brown University, 1989.

[14] W. Cook. Object-oriented programming versus abstract data types. In *Proceedings of the REX Workshop/School on the Foundations of Object-Oriented Languages*, volume 173 of *Lecture Notes in Computer Science*, 1990.

[15] B. C. d. S. Oliveira. Modular visitor components: A practical solution to the expression families problem. In S. Drossopoulou, editor, *23rd European Conference on Object Oriented Programming (ECOOP)*, 2009.

[16] O.-J. Dahl, B. Myhrhaug, and K. Nygaard. The SIMULA 67 common base language. Technical report, Norwegian Computing Center, 1970. Publication S-22.

[17] H.-D. Ehrich. On the theory of specification, implementation and parameterization of abstract data types. *J. ACM*, 29(1):206–227, 1982.

[18] A. Filinski. Declarative continuations and categorical duality. Master's thesis DIKU Report 89/11, University of Copenhagen, 1989.

[19] J. Gibbons. Unfolding abstract datatypes. In *MPC '08: Proceedings of the 9th international conference on Mathematics of Program Construction*, pages 110–133, 2008.

[20] J. Goguen, J. Thatcher, and E. Wagner. An initial algebra approach to the specification, correctness, and implementation of abstract data types. *Current Trends in Programming Methodology*, IV:80–149, 1978.

[21] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java(TM) Language Specification*. Addison-Wesley Professional, 2005.

[22] D. N. Gray, J. Hotchkiss, S. LaForge, A. Shalit, and T. Weinberg. Modern languages and Microsoft's Component Object Model. *Commun. ACM*, 41(5):55–65, 1998.

[23] C. A. Gunter and J. C. Mitchell, editors. *Theoretical aspects of object-oriented programming: types, semantics, and language design*. MIT Press, 1994.

[24] J. Guttag. *The Specification and Application to Programming of Abstract Data Types*. Report, University of Toronto, Computer Science Department, 1975.

[25] C. Hewitt, P. Bishop, I. Greif, B. Smith, T. Matson, and R. Steiger. Actor induction and meta-evaluation. In *POPL '73: Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 153–168. ACM, 1973.

[26] C. A. R. Hoare. Proof of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[27] J. Hogg, D. Lea, A. Wills, D. deChampeaux, and R. Holt. The Geneva convention on the treatment of object aliasing. *SIGPLAN OOPS Messenger*, 3(2):11–16, 1992.

[28] D. Ingalls. The Smalltalk-76 programming system. In *POPL*, pages 9–16, 1978.

[29] B. Jacobs. Objects and classes, co-algebraically. In *Object orientation with parallelism and persistence*, pages 83–103. 1996.

[30] S. P. Jones. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[31] S. P. Jones. Classes, Jim, but not as we know them. type classes in Haskell: what, why, and whither. ECOOP Keynote, 2009.

[32] B. W. Kernighan and D. Ritchie. *C Programming Language (2nd Edition)*. Prentice Hall PTR, 1988.

[33] S. Krishnamurthi, M. Felleisen, and D. P. Friedman. Synthesizing object-oriented and functional design to promote reuse. In *In European Conference on Object-Oriented Programming*, pages 91–113. Springer, 1998.

[34] B. Liskov. Keynote address - data abstraction and hierarchy. In *OOPSLA '87: Addendum to the Proceedings on Object-oriented programming systems, languages and applications (Addendum)*, pages 17–34, 1987.

[35] B. Liskov. A history of CLU. In *History of programming languages—II*, pages 471–510. ACM, 1996.

[36] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual*. Springer-Verlag, 1981.

[37] B. Liskov and S. Zilles. Programming with abstract data types. *SIGPLAN Notices*, 9(4):50–59, 1974.

[38] K. C. Louden. *Programming Languages: Principles and Practice*. Wadsworth Publ. Co., 1993.

[39] D. B. MacQueen. Modules for Standard ML. In *Conference on LISP and Functional Programming*, 1984.

[40] B. Mahr and J. Makowsky. An axiomatic approach to semantics of specification languages. In *Proceedings of the 6th Conference on Theoretical Computer Science*, volume 145 of *Lecture Notes in Computer Science*, pages 211–219. Springer-Verlag, 1983.

[41] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[42] R. Milner, M. Tofte, and R. Harper. *The definition of Standard ML*. MIT Press, 1990.

[43] J. C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, 2001.

[44] J. C. Mitchell and G. D. Plotkin. Abstract types have existential type. In *Proceedings of the ACM Symp. on Principles of Programming Languages*. ACM, 1985.

[45] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[46] D. A. Naumann. Observational purity and encapsulation. *Theor. Comput. Sci.*, 376(3):205–224, 2007.

[47] M. Odersky, L. Spoon, and B. Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 2008.

[48] M. Odersky and M. Zenger. Independently extensible solutions to the expression problem. In *Proceedings FOOL 12*, 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

[49] U. S. D. of Defense. Reference manual for the Ada programming language. ANSI/MIL-STD-1815 A, 1983.

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

[51] T. W. Pratt and M. V. Zelkowitz. *Programming languages: design and implementation*. Prentice-Hall, 1995.

[52] J. C. Reynolds. User-defined types and procedural data structures as complementary approaches to data abstraction. In *New Advances in Algorithmic Languages*, pages 157–168. INRIA, 1975.

[53] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.

[54] R. Sebesta. *Concepts of Programming Languages, Eighth Edition*. Addison-Wesley, 2007.

[55] J. F. Shoch. An overview of the programming language Smalltalk-72. *SIGPLAN Notices*, 14(9):64–73, 1979.

[56] G. Steele. LAMBDA: The ultimate declarative. Technical Report AIM-379, MIT AI LAB, 1976.

[57] A. B. Tucker and R. E. Noonan. *Programming Languages: Principles and Paradigms, Second Edition*. McGraw-Hill Higher Education, 2007.

[58] P. Wadler. The expression problem. Mail to the java-genericity mailing list, 1998.

[59] P. Wadler and S. Blott. How to make ad-hoc polymorphism less ad hoc. In *POPL '89: Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

[60] N. Wirth. *Programming in Modula-2*. Springer-Verlag, 1983.

[61] W. A. Wulf, R. L. London, and M. Shaw. An introduction to the construction and verification of Alphard programs. *IEEE Transactions on Software Engineering*, SE-24(4), 1976.

[62] S. N. Zilles. Procedural encapsulation: A linguistic protection mechanism. *SIGPLAN Notices*, 8(9):142–146, 1973.