Abstraction Mechanisms in Theta

Mark Day **Robert Gruber** **Barbara Liskov** **Andrew C. Myers**

Programming Methodology Group Memo 81
February 1994

MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA
{mday,gruber,liskov,andru}@lcs.mit.edu

Abstract

Abstraction mechanisms are crucial for the organization of large-scale software. Theta is a new object-oriented programming language developed for implementing shared persistent objects as components of large-scale software. Theta provides a number of mechanisms that support abstraction and the construction of large-scale software. No single mechanism is novel in itself. Instead, the value of Theta is the combination of relatively well-understood mechanisms into a coherent whole. Theta provides static type-checking with separate compilation; automatic storage management; specifications separate from implementation; subtyping; inheritance; and constrained parametric polymorphism. The mechanisms of inheritance, subtyping, and parametric polymorphism are entirely distinct in Theta. In contrast to other languages, constraints on parametric polymorphism are not expressed in terms of subtyping. We demonstrate that the use of subtyping to constrain polymorphism does not work properly. Our final contributions are to identify the problem of protecting classes from their inheriting subclasses, and to present rules that suffice for such protection.

1 Introduction

This paper describes a new object-oriented programming language called Theta. Theta was developed for implementing shared persistent objects as components of large-scale systems in the Thor object-oriented database system. In Thor, a paramount requirement is the secure, safe sharing of objects. Safe sharing requires that data abstraction be supported as fully as possible so that objects entrusted to the database can not be corrupted by other users. In Theta, abstractions are developed and implemented independently. Abstraction implementations are compiled separately, and programs are produced as needed by linking implementations together.

Theta has strong compile-time type checking; it guarantees that no type errors can occur after compilation (e.g., at link or run time). Theta objects reside in a heap and their storage is managed automatically. Strong type checking and automatic garbage collection together guarantee that programs are unable to violate encapsulation. Strong type checking has the further advantages that some common

This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136 and in part by the National Science Foundation under Grant CCR-8822158

errors are detected at compile time, and that more efficient implementations are sometimes possible because of the additional information available to the compiler. The possible disadvantage of strong type checking, namely difficulty in defining generic code, is lessened by providing generic capabilities more powerful than those in languages such as C++ and Modula-3.

Theta provides the following unique combination of features. First, it distinguishes specifications from implementations, and allows types to have multiple implementations even within the same program. Second, it provides support for both subtype and parametric polymorphism. Subtype polymorphism allows types to be organized in a hierarchy, where subtypes extend the behavior of their supertypes. Parametric polymorphism allows generic routines and types to be defined in terms of *parameter types*. Theta allows constraints on parameter types to be expressed in a way that is different from other object-oriented languages; we show later in this paper that the mechanisms in most object-oriented languages do not work properly. Third, Theta provides an inheritance (subclass) mechanism that allows new implementations to be based on old ones, but the inheritance mechanism is independent of the type hierarchy mechanism. Finally, it provides mechanisms that can prevent subclasses from causing any damage to superclass objects. Thus Theta provides complete encapsulation: both clients and inheritors are unable to interfere with the correct behavior of classes.

The remainder of this paper describes these mechanisms of Theta. In each case we discuss the issues that must be considered when designing the mechanism and the rationale for our decisions. We also discuss how the issues have been resolved in other object-oriented languages, and compare our approach with theirs. Our intention is not just to present Theta, but to provide an overview of the design issues and how they are treated across a number of languages.

We begin in Section 2 by explaining why we undertook the design of Theta, and the requirements we are trying to satisfy; we also discuss our design philosophy and the language design principles that we have attempted to follow. Section 3 describes our specification mechanism and how types and subtypes are declared. Then in Section 4 we discuss our parametric polymorphism mechanism. Section 5 describes implementations and inheritance. We conclude in Section 6 with a discussion of what we have accomplished.

2 Motivation

We undertook the design of Theta because of our research on Thor, a new object-oriented database system [11]. Thor supports heterogeneous sharing of persistent objects. Applications that use Thor can be written in different programming languages and a single application can have components in different programming languages sharing the common objects. Thor provides a universe of persistent objects. Each persistent object has an encapsulated implementation and a set of methods that can be used to interact with it. Each object also has a type that defines its methods; Thor provides a number of built-in types and users can define new types. Persistence in Thor is defined by reachability. The Thor universe has a persistent root; all objects accessible from the root are stored persistently and storage for objects that become inaccessible from the root is reclaimed automatically.

Since there can be many different users of Thor, it is important that they be protected from one another. Therefore, secure sharing is an important goal and Thor will provide access control mechanisms for this purpose. Most user code runs outside Thor and cannot interfere with the correct execution of the system (this point is discussed further in [4]). However, to define new Thor types, users must write code that runs inside Thor, and we needed a programming language to be used for this purpose. Security and safety were an absolute requirement for this language; an ill-behaved program must not be able to damage other users' objects or undermine the secure sharing mechanisms in any way.

Safe sharing first of all means that abstraction boundaries cannot be violated. Of course, good support for abstraction is important for control of complexity, module reuse, local reasoning, etc; in our environment, however, it also helps with secure sharing by ensuring that users share abstract objects rather than representations, and that they cannot interfere with one another by getting at encapsulated information. Proper support for abstraction depends on two things. First, abstractions must have specifications that describe what they do without prescribing how they are implemented. Second, implementations must be encapsulated so that other parts of the program cannot interfere with them. Without encapsulation it is not possible to reason

In addition, we needed a strongly-typed language. Thor provides type definitions that are used by programmers in developing the code that interacts with Thor. We didn't want those programmers to have to read code to understand the Thor objects they would be using. We also wanted the ability to change implementations without causing application code to break.

We considered using a widely-used language to implement Thor objects since this would make Thor more accessible. However, there is only one widely-used object-oriented language with static type checking, namely C++, and the type system of C++ is not secure. Since C++ wouldn't do, we decided to develop our own language, since this would allow us to learn what features were really needed. Interestingly, we have arrived at some of the same conclusions as work done on Portlandish [18], another experimental language designed for a distributed object store.

There are two design principles that guided the language design. We aimed for a design in which different concepts were expressed by different mechanisms because we believe that a language based on this principle is easier to understand than one that overloads a single mechanism for several different purposes. Also, we wanted the language to express "good" practice in a direct way. Although it is always possible to superimpose a good methodology on a language by defining conventions that programmers ought to follow, we prefer that a "good" idiom take a straightforward form. In both C++ and Modula-3, for example, the simplest ways of writing interfaces are not right for developing reliable large-scale software; ensuring appropriate information hiding requires the consistent use of extra-language apparatus.

3 Specifications and Type Hierarchy

We argued earlier that specifications are an intrinsic part of support for abstractions. Of course specifications can be distinguished from implementations without needing to have a special form for declaring them; this is the approach taken in both CLU [13] and Eiffel [15]. However, we decided that it would be best to have special forms for specifications in Theta; specifications are given separately from im-

```
bag = type

  % bags are multisets of integers

  put(val: int)
    % adds val to the bag

  get() returns (int) signals (empty)
    % removes and returns an arbitrary element of the bag; signals empty if none

  is_empty() returns (bool)
    % returns true if the bag is empty, else returns false

end bag
```

Figure 1: Specification of Bag

plementations. Here we are following the lead of a number of other languages: Modula-3 [17], ML [16], and Ada (and C++ [9] to some extent). Our specifications are "pure"; they contain no implementation detail, and therefore they do not constrain possible implementations. This is in contrast to C++, for example; in C++ programmers *can* define classes that contain no implementation detail and thus function as specifications, but they need not do so. However, not doing so can cause problems later: for example, when a second implementation is needed for the type, or when a subtype that differs in its implementation details is needed.

Theta has both types and routines. Thus, a routine independent of any class can truly stand alone, and need not be bundled into a fake class (as is necessary in Eiffel, for instance). A routine is a procedure or an iterator, and in either case can terminate normally (returning results in the case of a procedure) or by signaling an exception (possibly with some exception results). Both iterators and the exception mechanism are taken directly from CLU [13]. Types have objects with methods, and the methods can be procedures and iterators. In this paper we focus on types.

An example of a Theta type specification, for a bag type, is given in Figure 1. The specification identifies all the methods of the new type and gives their signatures. It also contains information about the type's behavior, but this information is written as comments, and is uninterpreted by Theta.

One point about these specifications is that they only define methods and there is no notion of "fields" associated with the type. We believe that having such a notion is confusing at best (since both clients and implementers tend to believe that a real field must correspond to the abstract one) and potentially damaging to abstraction: for example, the ODMG-93 standard [7] does not provide a way to specify a read-only field; Eiffel [15] allows a nullary method to be replaced by a field but not vice-versa.

One way in which Theta type specifications are unusual is that they *only* define the methods that can be invoked on objects of the type, and do not define any way of creating new objects of the type (or other "type operations" associated with the type itself). Type operations are omitted for two reasons. First, different implementations of a type may have different creators. For example, a hashed implementation of a bag might have a creator that takes in the hash function as an argument, while a sorted implementation does not need this information. Second, subtypes are likely to have creators that differ from those of the supertype.

Specifications for subtypes include a declaration of the type's supertypes; there can be many supertypes. The Theta compiler checks a subtype specification by comparing it with the specifications of each supertype: the subtype must have all methods of its supertype, and each method must have a signature that *conforms* to that of the associated method of the supertype [3]. A subtype specification is allowed to *rename* methods of the supertype; renaming is especially useful when a subtype has multiple supertypes because there may be name conflicts among the supertypes, and the renaming mechanism allows them to be resolved. The Theta renaming mechanism is broadly similar to those of Trellis [19] and Eiffel [15], although it differs in details.

Some object-oriented languages, such as Emerald [2], do not require an explicit declaration of the subtype-supertype relation, and instead infer the relationship based on method conformance. This syntactic approach to subtyping has two drawbacks. First, conformance of method signatures does not imply that the potential subtype meets the semantic requirements of the supertype [14]. Second, implicit subtyping rules out the possibility of method renaming. Implicit subtyping is useful when using subtyping to emulate parametric polymorphism (as discussed in Section 4.1), but the separation of the two mechanisms in Theta eliminates this rationale.

Figure 2 gives a specification of stack as a subtype of bag; the subtype extends the bag's behavior by providing two new methods, and it renames the bag methods put and get to push and pop, respectively. Note that these methods satisfy the type conformance rules, and that the subtype is meaningful because its objects behave like those of the supertype as far as any code using supertype methods can tell; a discussion of the subtype relation can be found in [14]. In this case the push and pop methods constrain the non-determinism of the corresponding put and get methods by indicating that pop returns the last element that was pushed.

We use the standard type conformance rule for the usual reasons. In Theta, any object referred to by variable v: T is guaranteed to belong to T or one of its subtypes. The compiler will allow all T methods to be called on v; the standard type conformance rule guarantees that these calls are legal, even if the method being called belongs to a subtype of T. The standard type conformance rule occasionally collides with a desire to make argument types in subtypes more specific (a feature provided in Eiffel); however, the requirement of safe, secure typing in a large system makes covariant argument types unworkable.

Theta allows more specific type information to be determined at runtime. For example, consider:

```
x: bag := y
...
h: int
typecase x
   when stack (z) => h := z.height( )
   others => h := 0
end
```

```
Stack = type bag {push for put, pop for get}

  push(val: int)
    % pushes val on top of the stack

  pop( ) returns (int) signals (empty)
    % removes and returns the top stack element; signals empty if none

  is_empty( ) returns (bool)
    % returns true if the stack is empty else returns false

  height( ) returns (int)
    % returns the height of the stack, i.e., the number of elements in it

  total ( ) returns (int)
    % returns the sum of the integers in the stack

end Stack
```

Figure 2: Specification of Stack

Within the typecase, if x actually refers to an object whose type is (a subtype of) stack, the arm labelled "stack" will be selected and z will refer to the object; within that arm, stack methods can be called on z. The typecase allows generic code to be written that will work on all subtypes of a supertype, but deal with particular subtypes differently when necessary. Modula-3 [17] has a similar typecase statement.

4 Parametric polymorphism

In this section we explain the need for constrained parametric polymorphism. This sort of polymorphism is not new: it was present in CLU [13], and extended with a form of renaming in Argus [10]. However, neither of those languages has subtype polymorphism. Some object-oriented languages have omitted parametric polymorphism entirely, and even those that include parametric polymorphism typically either provide no constraint mechanism or use subtyping for that mechanism. Theta has entirely distinct mechanisms for subtype polymorphism and constrained parametric polymorphism.

In this section, we provide a rationale for Theta's parametric polymorphism. We begin by examining the need for expressing constraints on type parameters. We then observe that subtyping is not sufficiently powerful for expressing many useful constraints. We contrast the constraint specification mechanisms of some existing languages, and describe the mechanism we have chosen for Theta. Since Theta has both parametric and subtype polymorphism, we define how they interact. Finally, we consider a novel feature of Theta: renaming and overriding at the point of instantiation.

4.1 The Need for Constraints

Consider a generic min_elt procedure that computes the minimum element in an array[T], for any type T. In Theta, a partial specification for this routine might look like:

```
min_elt[T] (x: array[T]) returns (T) signals(empty)
   % Requires: T has a less than or equal" method le that
   %           partially orders T elements
   % Effect: if x is empty, signals empty; otherwise
   %         returns the minimum element in x according to
   %         the partial order determined by T's le method
```

Here is a partial Theta implementation of such a procedure:

```
  % Note this is not legal Theta!
min_elt[T] (x: array[T]) returns (T) signals(empty)
  if (x.empty()) then signal empty end
  min: T := x.bottom()
  for elt: T in x.elements() do
     if elt ≤ min then min := elt end   % ≤ invokes le
     end
  return(min)
  end min_elt
```

This routine raises an exception if the array is empty, otherwise it computes the min element by iterating over the elements in the array. It assumes that it can invoke an le method (less than or equal) on the objects of type T in the array.

To use this routine, it must be *instantiated* with an *actual type*. For example, the instantiation min_elts[int] is a routine that can be used to compute the minimum integer in an array[int]. The instantiation min_elts[int] is equivalent to re-writing the above implementation, replacing "T" with "int" everywhere:

```
min_elt_for_int (x: array[int]) returns (int) signals(empty)
  if (x.empty()) then signal empty end
  min: int := x.bottom()
  for elt: int in x.elements() do
     if (elt.le(min)) then min := elt end
     end
  return(min)
  end min_elt_for_int
```

A compiler can type check this "rewritten" version of the routine as if it had been written by a programmer. The invocations of array methods empty, bottom, and elements are all correct (this will be true for any instantiation, since the correctness of the array method invocations does not depend on the parameter type). The invocation of the le method is also correct, since type int has an le method that takes an int and returns a bool. However, for some other instantiation min_elts[foo], this invocation may be incorrect, perhaps because foo does not have an le method.

Some languages use the "rewrite" approach that we just showed to type-check generic modules: at each instantiation point, the module is rewritten with the actual types of the instantiation and the rewritten version is type-checked. Both C++ templates and Modula-3 generics are checked in this way [9, 17]. Without a description of what is required of the parameter types, there is no contract between implementer and client. The client cannot tell what is required without reading the code, and has no guarantee that his code will continue to work with a different implementation of the abstraction.

Also, we consider it unacceptable that the compiler should have to check each instantiation of a parameterized module by instantiating the body of the module and then type-checking the resulting code. Indeed, it should be possible to type-check code that uses a parameterized module before that module has been implemented. To support this, one must capture the necessary type-checking information in the specification of the parameterized module. The result is *constrained* parametric polymorphism, where the specification of a parameterized module includes constraints on the legal instantiation types.

Here is a more complete Theta specification for min_elt:

```
min_elt[T] (x: array[T]) returns (T) signals(empty)
   where T has le(T) returns(bool)
   % Requires: T has a less than or equal" method le that
   %          partially orders T elements
   % Effect: if x is empty, signals empty; otherwise
   %        returns the minimum element in x according to
   %        the partial order determined by T's le method
```

We have added a *where clause*, used to specify constraints on type parameters; in this case, min_elt must be instantiated with a type that has an le method that orders T objects.

The Theta where clause is an adaptation of the CLU where clause [13]. In Theta, a where clause can specify any number of required methods for a type parameter, and these methods can then be used in the body of the parameterized module. If no required methods are given, no methods can be invoked. Unconstrained parameters are still useful for simple collection types or lookup tables where the elements are only stored and retrieved. For example, the built-in type array[T] does not require any methods for T elements.

Given a parameterized module with specified type parameter constraints, implementation and use of this module can be type-checked independently. At the point of use, the compiler checks that the actual instantiation type satisfies the constraints. For example, in the instantiation min_elt[int], the compiler checks whether actual type int satisfies the where clause of min_elt. This check works as follows: first rewrite the required method signatures from the where clause, replacing the parameter type(s) with the actual type(s); then verify that the method signatures of the actual type(s) conform to the rewritten method signatures. For min_elt[int], we check that int's le method conforms to the rewritten signature le(int) returns(bool) (which it does). Note that uses of min_elt can be checked even before an implementation has been provided.

An implementation of a parameterized module is checked just once. Invocations on objects of parameter type T are checked against the signatures of the required methods for T. In our example, the le method invocation in the generic min_elt above is correct when checked against the le signature given

```
sorted_collection = type[T]
    where T has le(T) returns(bool)
  % Requires: T has an le method that partially orders T objects.
  % What: A collection type that maintains the elements in sorted
  %       order based on the partial order defined by the le method.

 add(elt: T)
   % add elt to collection

 remove(elt: T)
   % remove elt from collection

 ...

 elements( ) yields (T)
   % yields the elements in sorted order, as determined by T's le method

end sorted_collection
```

Figure 3: Partial Specification of Sorted_Collection

in the where clause above. If the where clause were not provided, the compiler would report an error.

Although we are using parameterized routines in our examples, note that a very important use of constrained parameteric polymorphism is the definition of parameterized types. For example, Figure 3 shows a Theta specification for a collection type that keeps the elements in sorted order. This type can only be instantiated with an element type that provides an le method (to be used for ordering the elements).

4.2 Inadequacy of Types as Constraints

Trellis [19], Eiffel [15], Portlandish [18], and POOL [1] all have parameterized types, but use subtyping as the constraint mechanism: that is, they allow a parameterized abstraction to be instantiated with any type that is a subtype of the constraining type. However, a mechanism based on subtyping is not powerful enough to capture some useful constraints. The designers of the Emerald language initially chose a subtype-based mechanism but later realized its limitiations and changed to a type matching mechanism that is similar in power to Theta's where clauses; see [3] for a good discussion on this issue.

As an example, note that there is no way to rewrite min_elts so that it uses a "constraint type". To see this, consider this attempt:

```
    % This example is type-correct but useless

comparable = type
   le(comparable) returns(bool)
   end comparable
```

```
min_elt2(x: collection[comparable]) returns (comparable) signals(empty)
   if (x.empty()) then signal empty end
   min: comparable := x.bottom()
   for elt: comparable in x.elements() do
      if (elt.le(min)) then min := elt end
      end
   return(min)
   end min_elt2
```

Routine min_elt2 takes a collection[comparable], where type comparable has an le method. The code as given is type-correct, but it is useless: it cannot be used as intended (to compute the minimum element of any array whose objects have an le method).

Suppose we want to pass min_elt2 a collection[int]. This requires that int be a subtype of comparable.[1] Unfortunately, since we are using contravariance of argument types, int cannot be a subtype of comparable. Any subtype of comparable must have an le method that takes an argument of type comparable (or some *supertype* of comparable), while int has an le method that takes an int.

The comparable example actually works in Eiffel because the language allows covariant argument types. However, as mentioned above, such a covariance rule requires global program analysis to determine type correctness and seems ill-suited to the construction of large systems. Moreover, our main point is that the subtype relation should not be the only way to express constraints. Whether one has contravariance or covariance for argument types, there are constraints one can express with where clauses that one cannot express using simple type contraints.

4.3 Inadequacy of Conformance-based Approaches

Emerald uses types to express the same kinds of constraints that where clauses are used for in Theta [3]. For example, in Emerald we can define a type collection[comparable] where comparable has a definition similar to that in the previous example. However, Emerald does not use subtyping to determine whether a type is a valid parameter; instead, a collection[T] can be instantiated for any T that *matches* comparable, rather than just the T's that conform to comparable. When determining whether T matches comparable, we are allowed to substitute T for all occurrences of comparable in the specification of comparable, thus avoiding the covariance problem described earlier. This solution provides some of the power of where clauses. However, it introduces pseudo-types like comparable, which exist only to constrain other types; in all likelihood no objects satisfying the specification of comparable will ever be created. In Theta, such meaningless types are unnecessary.

F-bounded polymorphism [5] can describe many of the same useful polymorphic types. F-bounded quantification is essentially the same as the conformance-based approach used by Emerald, though it provides additional power if recursive types are allowed in a very general way. Theta's where clauses

[1]This example also requires that parameterized type collection be defined in such a way that if S is a subtype of T, collection[S] is a subtype of collection[T]. The issue of subtype relationships for parameterized types is discussed in Section 4.4.

are in practical terms more powerful than F-bounded polymorphism or Emerald conformance.

First, where clauses can introduce mutual dependencies between type parameters. For example, we can declare a method parameterized on types T and U such that T and U must be related by a particular set of methods. However, no subtype relation between T and U can be specified. Such mutual dependencies can be useful when designing methods whose signatures mention that take collections of two different types. Such mutual dependencies can be described (clumsily) by F-bounded quantification, but they require a very general

Second, where clauses allow the instantiator to override the methods that satisfy them, providing other methods or even stand-alone routines instead. Neither Emerald nor F-bounded quantification can directly express instantiation with overriding.

4.4 Combining Parametric and Subtype Polymorphism

In a language like Theta with mechanisms for both parametric and subtype polymorphism, we must define how they interact.

Suppose P1 and P2 are both parameterized types, with a single type parameter. When specifying P2, the following are two possible subtype declarations:

(1) P2[T] < P1[T] forall T
(2) P2[S] < P2[T] when S < T

Here, < is used to denote the *subtype-of* relationship. The first declaration says that for any type T, P2[T] is a subtype of P1[T]. The second declaration says: for any two types S and T, if S is a subtype of type T then P2[S] is a subtype of P2[T]. In a language that allows both forms, they can be used together to imply the following:

(3) P2[S] < P1[T] when S < T

Thus there is no need to support this third form directly.

The first form is useful and is supported in Theta. When specifying type P2 in Theta, one can give P1 as a supertype: the implication is that P2[T] is a subtype of P1[T] for all types T. The compiler verifies that P2's signature conforms to P1's; since the same type parameter T is used in both signatures, this is no harder than checking the conformance of two non-parameterized types. This supports the kind of type evolution we are interested in, e.g., introducing a subtype that adds an additional method or two to an existing type.

We believe the second form is less useful, and it is not currently supported in Theta. The reason is that such a declaration only works for types that do not use a parameter type as (a component of) the argument type of any method. For example, even if the form were supported, one could not declare

array[S] < array[T] when S < T *% This is wrong*

Consider the store method: for array[T] it takes an integer index and a T object, while for array[S] it takes an integer index and an S object. Contravariance of arguments requires that array[S]'s store method would take a T object, but clearly we cannot allow T objects to be stored in an array[S], since such an array is only supposed to contain objects of type S (or a subtype of S). Thus the contravariance of argument type rule correctly prevents us from making the above declaration.

All mutable collection objects with parameterized element type T have a mutation method such as store that takes a T object; for any mutable type M, we cannot declare M[S] < M[T] when S < T. What about immutable types? Most immutable types have at least one comparison method, and, like mutation methods, comparison methods have element type T in their signature, thus the same problem occurs. For example, the immutable type sequence[T] in Theta has an equal method that takes a sequence[T] as argument and returns a bool; thus, we cannot declare sequence[S] < sequence[T] when S < T.

We conclude that most real parameterized types (types that will have actual instances) cannot make use of the second form of subtype declaration. Note, however, that one can define "abstract supertypes" of these types that have no comparison or mutation methods. The simplest example is the following:

```
collection= type[T]
  elements() yields(T)
  end collection
```

Type collection captures the one behavior common to all collections parameterized by type T: the fact that they have T elements. We can declare that collection[S]˜$<$˜collection[T] when S˜$<$˜T, and this declaration works since no method arguments involve type T. One could then declare that types such as array[T] are subtypes of collection[T]. In this case one could write a routine draw_shapes that takes a collection[shape] and pass in an array[circle] (assuming circle˜$<$˜shape).

Thus, the second form of subtype declaration does have its uses, if one defines abstract supertypes such as collection. However, we believe most such abstract supertypes are not necessary; moreover, it will simplify the type hiearchy if they are not introduced. For example, the only information captured by type collection is the existence of an elements iterator, and we can use constrained parametric polymorphism to capture the same information:

```
draw_shapes[C](shapes: C, d: display)
    where C has elements() yields(shape)
  % C is a collection type that contains shapes
  for s: shape in shapes.elements() do s.draw(d) end
  end draw_shapes
```

We can then use this routine on an array[circle] by instantiating it with array[circle]. Since circle < shape, the elements method for array[circle] conforms to the one specified in the where clause, and the instantiation would be legal. This use of parametric polymorphism seems more direct than the use of the type collection, and has not introduced a new type declaration.

Of the languages we know of that have both subtype and parameteric polymorphism, the Trellis language [19] has the richest descriptive mechanism for expressing subtype relationships between parameterized entities. The initial Theta approach is very simple, and we are still exploring the question of whether it is sufficent.

4.5 Renaming and Overriding

When we write a where clause constraining a type parameter to have one or more methods, we choose names for these methods that are intended to convey the semantics required. However, it may be that different types use different names for a given method with the desired behavior; to handle this case, Theta supports renaming at the point of instantiation[2]

In some cases there can even be more than one appropriate method to use for a given required method. As an example of this, consider a min_elt[T] routine that requires an le method. Suppose type color has methods darker and lighter. We can compute the lightest and darkest colors in an array of colors as follows:

```
darkest := min_elt[color {darker for le}](mycolors)
lightest := min_elt[color {lighter for le}](mycolors)
```

Sometimes a type does not have a required method, but the necessary functionality can be provided by some other routine. Theta supports this with procedure overriding at instantiation time. For example, suppose type color only has a brightness method, but no darker or lighter methods. We could write the following:

```
darker(x, y: color) returns(bool)
  return (x.brightness() < y.brightness())
  end darker

darkest := min_elt[color {op darker for le}](mycolors)
```

In this case the keyword op indicates that a stand-alone routine, rather than a method, should be used in place of le.

Suppose the required method for type parameter T is foo(at1, at2, ..., atk) returns(rt), where at1 through atk are argument types and rt is the result type. The supplied stand-alone routine must have a signature that conforms with the signature (T, at1, at2, ..., atk) returns(rt). This allows the supplied stand-alone routine to be used wherever the required method is invoked within the parameterized module.

We believe that Theta is unique in its mechanisms for renaming and overriding at the instantiation point.

5 Classes and Inheritance

A type is implemented by a unit of code that, following standard terminology, we call a *class*. A class usually implements a type extension. Such a class implements both the operations defined in an extension and the methods of the type it extends. (A class used purely for code sharing by subclasses, corresponding roughly to an abstract class in C++ or a deferred class in Eiffel, may implement only a type). There can be many different classes implementing an extension, and we do not express the "implements" relation by using the subtype relation, as is done in some languages. The main reason for

[2] The Theta mechanism is an extension of a renaming mechanism that was designed for Argus [10] but never implemented.

not doing this is that it exposes implementations too much, and allows code to become dependent on an implementation when it ought to be dependent only on the type being implemented.

Classes in Theta are similar to those in other languages. The class defines a set of instance variables, and provides implementations of the methods and operations in terms of those variables. The class must implement all the methods and operations defined by the type and extension, and in addition can implement some private methods and operations. The signatures of the public methods and operations must conform to those in the extension and type specifications.

A class can be defined as a subclass of some other class, using Theta's inheritance mechanism. Inheritance provides several advantages. It allows the programmer to conveniently reuse code when implementing a subtype; it allows the creation of abstract classes, where a partial implementation provides a template that is filled in by subclasses; and it provides efficient sharing of code.

The Theta inheritance mechanism is used only for sharing implementation details among classes. Because it is distinct from the subtype mechanism, inheritance can be used for implementations of unrelated types. For example, suppose C1 is a subclass of C2, where C1 implements type T1 and C2 implements type T2. We do not require that T1 be a subtype of T2, although this will sometimes be the case. As others have pointed out [20, 8, 6], it is a good idea to keep these hierarchies distinct, since they are solving different problems. The type hierarchy is concerned with behavior, and subtypes of the same type may be implemented very different. The inheritance mechanism is concerned with implementation, and it is sometimes useful to borrow an implementation of an unrelated type. Some other languages (POOL [1], Portlandish [18]) have similarly chosen to separate subtyping and inheritance.

Theta supports only single inheritance. We believe this is sufficient because we use inheritance only for code sharing. Languages such as C++ and Eiffel use multiple inheritance in several ways: code sharing, type hierarchy, multiple implementations, and as a way of achieving or constraining parametric polymorphism. When inheritance is used for several things, a class often needs to have several superclasses, some to express type hierarchy, some to express the "implements" relation, some to express parametric polymorphism, and perhaps some others for code sharing. However, in Theta we use inheritance only for code sharing, and we believe that a single superclass will be sufficient. Avoiding multiple superclasses is good because we thereby avoid the complexity associated with resolving conflicts among multiply-inherited instance variables, as well as being able to avoid the cost of a multiple-inheritance method dispatch mechanism.

Figure 4 shows a class that implements stacks (as specified in Figure 2) by inheriting from the Theta built-in array class. Stacks are not a subtype of arrays, but it is convenient to base their implementation on arrays, since arrays in Theta can grow and shrink using the `addh` and `remh` operations.

The figure illustrates some features of Theta classes. First, the subclass can rename superclass methods; here we rename `size` to `height` since it does exactly what is wanted. Second, note the use of up-arrow to name superclass methods and operations, e.g., `^create` identifies the array create operation as opposed to the stack create operation.

As is usual, the subclass can inherit and override methods of the superclass. Methods that it overrides must conform to the signatures provided in the superclass. The reason for this is that they may be called in other superclass methods; conformance guarantees that those calls will be legal.

```
s = stack class inherits array[int] {height for size}

  sum: int

  push (x: int)
    sum := sum + x
    ^addh(x)          %  array.addh appends one element to the end
    end push

  pop ( ) returns (int) signals (empty)
    x: int := ^remh()  %  array.remh removes the last element
      except when bounds: signal empty end
    sum := sum - x
    return (x)
  end pop

  is_empty ( ) returns (bool)
    return (^size() = 0)
  end is_empty

  total ( ) returns (int)
    return (sum)

end s

create makes (s)
  init {sum := 0} ^create() end
end create
```

Figure 4: Implementation of stack

5.1 Makers

Figure 4 also illustrates the use of a *maker*. Some operations and methods that return new objects are implemented by having them create the new objects themselves while others are implemented by making a call to create the new object. We consider this choice to be an implementation detail that should not be visible in the specification. Inside the class, methods and operations that create the new object directly are implemented as makers. A maker has an additional implicit argument, the new object that it is initializing; this object can be referred to (using the name newobject), but only after its fields have been filled in by the init statement. (In the example the maker returns immediately after filling in the fields.)

A maker in a subclass initializes the superclass fields by using a maker of the superclass within the init statement. The superclass maker is called only after the subclass fields have been initialized; this guarantees that the new object is accessible outside the maker only when all its fields are initialized.

C++ constructors are similar to Theta makers, but initialize in the reverse order: the subclass fields are initialized only after the superclass portion of the object is filled in. Because the subclass fields are uninitialized during superclass initialization, any method invocations must call the superclass versions, since the subclass methods would depend on uninitialized fields. This effect has a runtime cost, since C++ constructors overwrite the object dispatch information at each stage of object initialization. Theta objects, on the other hand, can be initialized immediately with the proper dispatch information.

In addition to type extension operations, Theta makers can also be used to directly implement methods. Thus a copy method can be implemented as a maker in Theta, whereas it must invoke a constructor in C++.

5.2 Encapsulation

In most object-oriented languages, object implementations are encapsulated so as to be inaccessible to client modules, but they often remain vulnerable to subclasses. In Theta it is possible to implement classes in a way that ensures that subclasses cannot interfere with objects of the class.

First, subclasses are not allowed access to superclass fields. Each subclass object contains the fields defined by the superclass: the object's representation (rep) is a concatenation of the superclass fields and the subclass fields. But within the subclass the superclass fields cannot be accessed directly; instead they can be accessed only by the methods available to the subclass.

Limiting access in this way means that the subclass can cause the superclass fields to violate the superclass *rep invariant* [12] only if the superclass provides methods and operations that do not ensure the invariant (i.e., they terminate at a point when the superclass fields do not satisfy the invariant). This limitation makes it easier to implement the subclass correctly. It also protects the superclass: if all inherited fields of subclass objects satisfy the superclass rep invariant, then there is no danger that superclass methods will propagate bad information into superclass objects.

For example, suppose that stack has a copy method, and also suppose the s implementation shown earlier has a private set_sum method and implements these methods by

```
maker copy ( )
  init {sum := self.sum} ^copy() end
  end copy

set_sum (x: int)
  sum := x
  end set_sum
```

The set_sum method is a "violator" because it does not ensure the rep invariant of s (the invariant being that sum is equal to the sum of the array elements). If this method is not exported to subclasses of s, they cannot cause a violation of s's rep invariant for the s fields of their objects. If the method is exported, however, such violations are possible. Of course, the correctness of s only concerns s objects. However, if the copy method is also exported to subclasses, it can "propagate" the bad information to an s object: when it is called on a subclass object it just copies the fields without checking the rep invariant, and the result may be an invalid s object.

A class that exports only public methods and operations to its subclasses cannot be hurt by them. A class can also export private methods and operations and there will be no problem provided these also ensure the rep invariant. If the class exports "violator" methods and operations, there still will be no problem provided it does not also export "propagator" methods like copy to its subclasses

To provide the needed power, we allow a class to control the interface it exports to its subclasses. The default is that all public methods are exported and all private methods are hidden. The following is an example of overriding the default:

```
s = stackE class inherits array[int] {height for size}
      exports set_sum    % used to export a private method
      hides copy         % used to hide a public method

   ...
end s
```

There is one final problem with inheritance in Theta: we need to be sure that a subclass object cannot masquerade as an object of the type implemented by the superclass[3]. This problem can only arise for classes with trivial implementations of copy-like methods. For example, to implement copy for sequences (immutable arrays), we might just write

```
copy ( ) returns (seq)
  return (self)
  end copy
```

Here copy returns the very same object; this is legitimate for immutable types because the sharing isn't visible to clients. However, if copy is inherited, the result when it is called on a subclass object is to make that object appear to be an object of type seq. To avoid such a problem, a class with such methods simply avoids exporting them to its subclasses.

6 Conclusion

This paper has described Theta, a new object-oriented programming language. Theta was developed for use in an environment in which secure, safe sharing was of paramount importance. This led to a number of decisions about the language: complete static type checking with separate compilation, automatic storage management, and specifications separate from implementations.

Theta provides a combination of features that makes it different from other languages; in addition, some of the features are themselves novel. Of most interest, we believe, are the separation of types from extensions, the separation of inheritance from subtyping, the way that type constraints for parametric

[3]In theory this would not be a problem in a language in which the two hierarchies were the same, although in practice it might very well be a problem because some subclasses might have objects that did not behave like those of the superclass' type.

polymorphism are expressed, the construction of new objects by makers, and the mechanisms that allow classes to be protected from their subclasses.

The paper has also discussed the issues that arise in designing a language like Theta. We have argued that parametric and subtype polymorphism are distinct and ought to be supported by separate mechanisms. In addition, we showed that constraints for parametric polymorphism cannot be expressed adequately using a subtype mechanism.

An early version of Theta is in use as the interface specification language for our Thor prototype implementation; that version distinguishes subtyping from inheritance but does not include any of the other features described in this paper. We are currently implementing the full language, including the features described here.

References

[1] Pierre America and Frank van der Linden. A parallel object-oriented language with inheritance and subtyping. In *ECOOP/OOPSLA '90 Proceedings*, pages 21–25, October 1990.

[2] Andrew Black, Norman Hutchinson, Eric Jul, Henry Levy, and Larry Carter. Distribution and abstract types in Emerald. *IEEE Transactions on Software Engineering*, SE-13(1):65–76, January 1987.

[3] Andrew P. Black and Norman Hutchinson. *Typechecking polymorphism in Emerald*. Technical Report 91/1, Digital Equipment Corporation, Cambridge Research Laboratory, December 1990.

[4] Philip Bogle and Barbara Liskov. Reducing cross-domain call overhead using batched futures. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages ???–???, 1994.

[5] Peter Canning, William Cook, Walter Hill, John Mitchell, and Walter Olthoff. F-bounded polymorphism for object-oriented programming. In *Proceedings of the Conference on Functional Programming Languages and Computer Architecture*, pages 273–280, 1989.

[6] Peter S. Canning, William R. Cook, Walter L. Hill, and Walter G. Olthoff. Interfaces for strongly-typed object-oriented programming. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 457–467, 1989.

[7] Richard G.G. Cattell, editor. *The Object Database Standard: ODMG-93*. Morgan Kaufmann, 1994.

[8] William R. Cook. Interfaces and specifications for the Smalltalk-80 collection classes. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 1–15, 1992.

[9] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[10] Barbara Liskov, Mark Day, Maurice Herlihy, Paul Johnson, Gary Leavens, Robert Scheifler, and William Weihl. *Argus Reference Manual*. Technical Report 400, MIT Laboratory for Computer Science, November 1987.

[11] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In *Distributed Object Management*. Morgan Kaufmann, 1994.

[12] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[13] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *CACM*, 20(8):564–576, August 1977.

[14] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, November 1994.

[15] Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, 1992.

[16] Robin Milner, Mads Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, Cambridge, MA, 1990.

[17] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[18] Harry H. Porter, III. Separating the subtype hierarchy from the inheritance of implementation. *Journal of Object-Oriented Programming*, pages 20–29, February 1992.

[19] Craig Schaffert, Topher Cooper, and Carrie Wilpolt. *Trellis Object-Based Environment, Language Reference Manual*. Technical Report DEC-TR-372, Digital Equipment Corporation, November 1985. Published as *SIGPLAN Notices 21(11)*, November, 1986.

[20] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *ACM Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 38–45, 1986.