Subtypes vs. Where Clauses: Constraining Parametric Polymorphism

Mark Day

Lotus Development Corporation
1 Rogers Street
Cambridge, MA 02142
Mark_Day@crd.lotus.com

Robert Gruber Barbara Liskov Andrew C. Myers

Laboratory for Computer Science,
Massachusetts Institute of Technology,
545 Technology Square, Cambridge, MA 02139
{gruber,liskov,andru}@lcs.mit.edu

Abstract

All object-oriented languages provide support for subtype polymorphism, which allows the writing of generic code that works for families of related types. There is also a need, however, to write code that is generic across types that have no real family relationship. To satisfy this need a programming language must provide a mechanism for parametric polymorphism, allowing for types as parameters to routines and types. We show that to support modular programming and separate compilation there must be a mechanism for constraining the actual parameters of the routine or type. We describe a simple and powerful constraint mechanism and compare it with constraint mechanisms in other languages in terms of both ease of use and semantic expressiveness. We also discuss the interaction between subtype and parametric polymorphism: we discuss the subtype relations that can exist between instantiations of parameterized types, and which of those relations are useful and can be implemented efficiently. We illustrate our points using examples in Theta, a new object-oriented language, and we describe the time- and space-efficient implementation of parametric polymorphism used in Theta.

Web URL: http://www.pmg.lcs.mit.edu/. 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.

1 Introduction

Good software engineering practice encourages the writing of generic code, which captures a common behavior and is usable in multiple contexts. The writing of generic code is partially supported by subtype polymorphism. With subtype polymorphism, one can define a family of related types: a supertype defines the behavior common to all its subtypes, which extend or specialize that behavior. Programs are then generic with respect to the defined family, since code written in terms of an object of type T is also usable for an object whose type S is a subtype of T.

However, there is also a need for code that is generic across types that have no real family relationship. For example, aggregates like lists and hash tables are typically useful for a wide variety of contained types, none of which is necessarily a subtype of any other. In such cases, the construction of generic code depends on parametric polymorphism. Parametric polymorphism allows the programmer to write code whose typing is incomplete in a specific way: it has one or more type parameters. This code is effectively a template that expresses a generic behavior, abstracted over the types with which that behavior can be used.

All object-oriented languages have well-developed mechanisms for subtype polymorphism, but in most of them the support for parametric polymorphism is either absent or too weak. The paper discusses how to provide both kinds of polymorphism in a strongly-typed language with compile-time type checking and separate compilation. Its main contribution is to bring together in one place a discussion of all of the relevant design issues. The paper makes the following points:

1. It shows that support for abstraction and separate compilation requires a compile-time enforceable

mechanism for constraining the instantiations of a parameterized abstraction.

2. It describes a constraint mechanism, *where clauses*, that is both simple and powerful, and contrasts where clauses with other constraint mechanisms in terms of both ease of use and semantic power.

3. It discusses the distinction between derived subtyping and declared subtyping, and explains the advantages of declared subtyping.

4. It discusses how to combine subtype and parametric polymorphism. It explains what subtype relationships can hold between parameterized types, and discusses how these relationships interact with implementation concerns.

The discussion is illustrated with examples from a new object-oriented language called Theta. We use the Theta design as an example of a coherent set of design decisions that provide effective support for both subtype and parametric polymorphism. We present Theta's encapsulation mechanism and discuss the relationship of encapsulation to polymorphism. We also describe our time- and space-efficient implementation of parametric polymorphism. The paper considers only a portion of Theta. A complete description can be found in [LCD$^+$94].

The remainder of the paper is organized as follows. We begin by discussing types and subtype polymorphism in a strongly-typed programming language. Section 3 discusses parametric polymorphism and mechanisms for constraining type parameters, while Section 4 discusses subtype relations among instantiations of parameterized types. Section 5 describes modules and encapsulation in Theta, and Section 6 describes our implementation of parametric polymorphism. We conclude with a discussion of what we have accomplished.

2 Types and Subtypes

The goal of subtype polymorphism is to support code that is generic with respect to a family of related types. Code written in terms of some type T actually works for all subtypes of T. The supertype defines the behavior common to all its subtypes, which then extend or specialize that behavior. For example, if stack were a subtype of bag, all code that worked for bags would also work for stacks.

A strongly-typed object-oriented language requires explicit definitions of type interfaces so that the compiler can insure that all calls of methods are legal. The compiler also needs to know the *type hierarchy*: which types are subtypes of which other types. In Theta this information is provided in type *specifications*. A type specification lists the methods of a type's objects and their signatures; it may also declare one or more supertypes. Specifications only describe interface information and do not contain any code that implements the type being defined. We chose this approach because it mirrors the distinction between a type's behavior and implementations of that behavior: the specification serves as a contract that its users rely on and its implementors must support. Substitution of one correct implementation for another does not compromise the correctness of using code, and, in fact, Theta allows multiple implementations of the same type to be used within a single program.

Figure 1 gives examples of Theta type specifications. Both string_set and string_bag are collections of strings; however, if a string is added to a string_bag several times it appears in the bag multiple times (and the elements iterator will yield it many times) whereas it would appear in the set just once. (An iterator is a special kind of routine that *yields* a sequence of results one at a time [LSAS77].) The specification for string_int_map indicates that it is a subtype of string_set (using the $<$ symbol); this makes sense because string_int_map stores a single mapping for each string, and defines the elements iterator to yield the strings that are mapped. The specification of string_int_map *renames* some string_set methods: insert is renamed insert_default, and elements is renamed keys. Renaming allows a subtype to establish an arbitrary correspondence between its method names and those of its supertypes; it is especially useful when there are multiple supertypes because it allows the subtype specification to resolve name conflicts, e.g., if two supertypes use the same name for methods with different behavior.

The Theta compiler checks each subtype declaration for legality. A subtype has all the methods of its supertypes; any method specifications not explicitly in the subtype specification are obtained from its supertype(s), applying renamings appropriately. (We believe it is good practice to include declarations of

```
string_set = type
  insert (x: string)
    % Adds x to self if it is not already there.
  remove (x: string) signals (not_found)
    % Removes x from self if it is a member;
    % otherwise, signals not_found.
  elements ( ) yields (string)
    % Yields every element of self, each exactly once,
    % in arbitrary order.
end string_set

string_bag = type
  insert (x: string)
    % Adds x to self; if x was already in self, it now
    % has one additional entry in self
  remove (x: string) signals (not_found)
    % Removes one occurrence of x from self if it is a member;
    % otherwise, signals not_found.
  elements ( ) yields (string)
    % Yields every occurrence of every element of self,
    % each exactly once, in arbitrary order.
end string_set

string_int_map = type < string_set
      {insert_default for insert, keys for elements}
  insert (key: string, value: int)
    % Adds a mapping from key to value,
    % replacing the previous mapping, if any.
  insert_default (key: string)
    % Adds a mapping from key to a default value,
    % replacing the previous mapping, if any.
  keys ( ) yields (string)
    % Yields every key in the map, each exactly once,
    % in arbitrary order.
  fetch (key: string) returns (int) signals (not_found)
    % Returns the value that key is mapped to. If it is
    % not mapped to anything, signals not_found.
end string_int_map
```

Figure 1: Some Theta Type Specifications

inherited methods if the behavior has changed — this is why we include specifications of the insert_default and keys methods in the specification of string_int_map.) Thus for each method m_t of supertype T, there is a corresponding method m_s in S. The subtype declaration is *legal* iff the signature of each m_s *conforms* to that of the corresponding m_t. The signature of m_s conforms to that of m_t iff:

- *Contravariance of arguments*: m_s has the same number of arguments as m_t, and the type of each argument of m_s is a *supertype* of the type of the corresponding argument of m_t.
- *Covariance of results*: m_s has the same number of results as m_t and the type of each result is a *subtype* of the type of the corresponding result of m_t.
- *Covariance of exceptions*: The exceptions of m_s are a subset of those of m_t, and the results for each exception must be covariant.

This is the usual definition of conformance [SCW85, BHJ$^+$87, Car84] extended to support exceptions. As discussed by others, e.g., [BHJ$^+$87], the first two rules are the weakest rules that ensure that static type checking guarantees no run-time type errors. The rule for exceptions ensures that a subtype method signals only exceptions defined for the supertype method. Therefore, a caller using the supertype interface receives only exceptions that are declared in that interface.

All relations among user-defined types are *declared explicitly* in Theta. Each type specification defines a unique type that is not equal to the type introduced by any other Theta specification. Thus string_set and string_bag are not the same type. Similarly, one type is a subtype of another only if its specification declares this *and* the declaration is legal. In some object-oriented languages, e.g., Emerald [BHJ$^+$87], PolyTOIL [BSvG95], and School [RIR93], type relations are *derived*: two specifications define the same type if they are identical except for the names of the types they are defining, and a type is a subtype of another type if it provides the necessary methods and their signatures conform. Declared type relations provide a finer granularity of type checking than is possible when type relations are derived; also, they allow renaming of methods to resolve conflicts among supertype methods, and can lead to more efficient language implementations [Mye95].

For example, string_set objects behave differently from string_bag objects even though they have the same methods. In Theta these two types are distinct, and

```
create_sorted_string_set ( ) returns (string_bag)
  % Returns a new, empty set with a sorted
  % implementation.

create_string_set ( ) returns (string_set)
  % Returns a new, empty set.

create_hashed_string_map (default: int, size_hint: int)
                        returns (string_int_map)
  % Creates a new, empty string_int_map whose
  % default value is default. Uses a hash table
  % implementation. The argument size_hint hints at the
  % number of mappings ultimately in the map.
```

Figure 2: Specifications of Creators

neither is a subtype of the other, whereas in languages with derived type relations, they would be viewed as the same type, and each would be a subtype of the other. Similarly, string_int_map is not a subtype of string_bag since a string_bag keeps track of duplicates whereas string_int_map does not. (See [LW94] for further discussion of when subtyping makes sense behaviorally.)

Type specifications in Theta define only the methods of the type's objects but do not define ways to create objects "from scratch." Such "creators" are specified separately and are typically stand-alone routines (i.e., not methods). Some example creator specifications are given in Figure 2. There are two reasons for separating types from creators:

1. Different implementations of a type may need different creators, e.g., create_hashed_string_map takes a hash-table-specific argument.

2. Subtypes sometimes need different creators from their supertypes.

Type and routine specifications are implemented in *modules*; some examples are given in Section 5.

3 Parametric Polymorphism

Sometimes it is useful to write generic code that is polymorphic with respect to types that are not in the same family. For example, searching an array for a match with a given element is useful over a wide range of element types. However, it doesn't make sense for *all* element types because there has to be a way of testing for a match (such as calling a method that determines equality). As another example, we might want to sum the elements of any collection type (e.g., int_set, int_bag, int_stack) that allowed us to access all its elements by means of an elements method. In general, only types that satisfy certain "constraints" are allowed, where the constraints rule out types whose objects don't have certain methods (*i.e.*, we need "constrained genericity" [Mey86]).

One might try to capture constraints using the subtype relation. Consider a type comparable that is intended to describe all types that have a method that determines whether one object is greater than another:

```
comparable = type
    gt(x: comparable) returns (bool)
        % returns true if x > self else returns false
end comparable
```

Seemingly, such a type could be used to express the signature of a sort routine:

```
sort(a: array[comparable])
```

We would then like to call the routine with, say, an array[int]; the code would call the gt method for array elements.

This approach does not work, for two reasons. First, array[int] is not a subtype of array[comparable] even if int $<$ comparable; we discuss this issue further in Section 4. Second, there are essentially no useful subtypes of comparable, because the contravariance of arguments in the method conformance rules has undesirable consequences [BHJ+87, RIR93]. A subtype of comparable must have a gt method, and (because of contravariance), the argument of this method must have type comparable, or a supertype of comparable! Thus, for int to be a subtype of comparable, the signature of the int gt method must be:

```
gt (x: comparable) returns (bool)
```

In general, an int and a comparable cannot be compared. An implementation of this gt method for type int would first attempt to cast the argument down to an int, and if this succeeded it would do the comparison. The cast is possible in Theta using the *typecase* statement; it

```
set = type[T] where T has equal(T) returns (bool)
  insert(x: T)
  remove(x: T) signals (not_found)
  elements( ) yields (T)
  equal(s: set[T]) returns (bool)
end set

sort [T] (a: array[T]) where T has gt (T) returns (bool)
  % sorts a into increasing order based
  % on the gt method of its elements
```

Figure 3: Parameterized Specifications

requires a run-time check. Clearly we do not want to follow this path, as it effectively eliminates an important part of compile-time type-checking for gt. Furthermore, forcing all implementations of subtypes of comparable to handle arbitrary comparable objects is both a nuisance and costly at run-time.

It is worth noting that Meyer [Mey86] concluded that the above problem with comparable did not arise, but he was using an unsound definition of conformance that allowed covariant argument types [Coo89].

3.1 Expressing Constraints

The Theta mechanism for supporting genericity is constrained parametric polymorphism, which was first introduced in CLU [LSAS77]. With this mechanism, types are explicitly provided as parameters to polymorphic types and routines. A parameterized definition defines a set of types or routines; the set contains an element for each legal substitution of actual types for the type parameters. To select a type (or routine) from the set, we *instantiate* the definition, providing an actual type for each of its parameters. For example, Figure 3 contains some parameterized specifications; examples of legal instantiations are the types set[int], and set[set[int]] and the routine sort[char]. In the case of a routine, instantiation typically happens in conjunction with a call, e.g., sort[char](ca), but it is possible to separate these activities: e.g, to do an instantiation to obtain a routine that is stored in a data structure and called later.

Having explicit type parameters isn't the whole story, because we still need a way of expressing constraints on allowable parameters. This is the purpose of the *where clause*. Originally defined in CLU [LSAS77], the where clause has been adapted in Theta (and also in School [RIR93]) to an object-oriented language with subtyping. A where clause lists the names and signatures of required methods for the parameters. For example, the where clause in the specification of set indicates that any legal parameter must have an equal method.

Where clauses allow the compiler to type check instantiations and implementations independently; thus this approach supports separate compilation. Intuitively, an instantiation is legal if the actual type has the methods listed in the where clause; an implementation is legal if, for objects of a parameter type, it only uses the methods listed in the where clause. If no required methods are listed for a parameter, no methods can be invoked on objects of that type. Unconstrained parameters are useful for simple collection types or lookup tables where the elements are only stored and retrieved. For example, the built-in parameterized type array[T] does not require any methods for T elements. (Theta allows methods to have where clauses of their own that impose additional constraints on the parameters of their type. For example, the array copy method requires that T have a copy method.)

Now we define our legality checks in more detail. To check the legality of an instantiation, the actual types are substituted for their associated parameters in the signatures given in the where clause. Then, for each method name in the where clause for a parameter, the actual type associated with that parameter must have a method of that name with a signature that conforms to the signature in the substituted where clause. For example, sort can be instantiated with any type foo having a gt method whose signature conforms to proc (foo) returns (bool); thus, sort[int] is allowed but sort[set[int]] is not (since set[int] does not have a gt method). The result of the instantiation is a type or routine signature obtained by substituting the actual types for the parameters and removing the where clause. Thus, sort[char] has the signature proc (array[char]).

To check the legality of parameterized definitions the compiler proceeds in the normal way, except that it assumes the existence of a type for each parameter, with methods as specified in the where clause. These parameter types have no subtypes and no supertypes (except for type any, which is the supertype of all types in Theta). For example, suppose the following code were within an implementation of sort:

```
sort [T] (a: array[T]) where T has gt (T) returns (bool)
  ...
  x, y: T
  if x > y ...     % legal – T has gt
  x := x.incr( )   % not legal – T does not have incr
  z: array[T]      % legal
  s: set[T]        % not legal – T does not have equal
```

The if statement is legal because T objects have a gt method (> is a special form for calling this method), and the type of y is a subtype of gt's argument type; the call x.incr is not legal because T objects do not have an incr method. The instantiation array[T] is legal because array does not require any T methods; the instantiation set[T] is not legal because it requires an equal method for its parameter, and T doesn't have this method.

Where clauses can introduce dependencies between type parameters. For example, we could define a member routine that works on arbitrary indexed collections:

```
member[E, C](a: C, x: E) returns (bool)
   where E has equal(E) returns (bool),
      C has elements( ) yields (E)
```

(Elements is an iterator; recall that an iterator returns a sequence of results one at a time.) Here the constraint for type C, a collection type, depends on type E, the type of the elements stored in C. An instantiation on multiple parameters is legal when the methods of all the actual types conform to the where clause declarations with all substitutions having been performed on each where clause. Thus the instantiation member[int,set[int]] is legal but member[char,set[int]] is not.

(1) Theta (where clauses):

```
member[E, C](a: C, x: E) returns (bool)
   where E has equal(E) returns (bool),
         C has elements( ) yields (E)
```

(2) Rapide (parameterized subtype constraints):

```
type HasEql(type T) is
  interface
  equal: function(x: T) return Boolean;
  end interface

type HasElts(type T) is
  interface
  elements: iterator( ) yield T;
  end interface

member: function(type E <: HasEql(E),
                 type C <: HasElts(E),
                 a: C, x: E) return Boolean;
```

(3) Emerald (type matching clauses):

```
% Emerald has no stand-alone routines, thus we define
% an object type with a polymorphic member method:

const MemberObjType ← typeobject
  function member [E: type, C: type, a: C, x: E]
        → [Boolean]
    suchthat E ⊳ typeobject EqAble
        function equal [EqAble] → [Boolean]
      end EqAble
    suchthat C ⊳ typeobject HasElts
        iterator elements[ ] ⇒ [E]
      end HasElts
end MemberObjType
```

Figure 4: Three Ways to Specify the member Routine

3.2 Discussion

Some languages with parametric polymorphism do not have a mechanism for stating constraints on type parameters, e.g., C++ templates [ES90] and Modula-3 generics [Nel91]. To check that an actual instantiation of a parameterized routine or type is correct, the compiler rewrites the body of the routine or type, replacing the type parameter with the actual type, and then checks the result. This approach violates abstraction and modularity, since there is no way to isolate constraints so that users can depend on them independently of particular implementations. Changing an implementation can break code that uses the abstraction being implemented. In addition, the approach does not support separate compilation, since the body of the instantiated type or routine must be available to the compiler.

Some languages use type definitions rather than where clauses to express type constraints. Two variants of this approach are shown in Figure 4. This figure shows the different ways of specifying the member routine discussed earlier; it repeats the where clause example, followed by examples in the languages Rapide [KLMM94] and Emerald [BHJ+87]. (Rapide and Emerald do not have iterators, so we had to invent some syntax.)

To constrain a type in Rapide one defines an auxiliary parameterized type to capture the desired constraint (the desired set of methods). In this case, the parameterized type HasEql(type T) captures the need to constrain a type to have an equal method that takes a T argument and returns a boolean, while HasElts(type T) captures the need to constrain a type to have an elements iterator that yields T objects. These auxiliary types are used in the specification of member: parameter E is required to be a subtype of HasEql(E) (written E <: HasEql(E)), i.e., E must have an equal method that takes an E. Similarly, parameter C is required to be a subtype of HasElts(E).

Emerald expresses constraints by introducing a new relation between types called *matching*. E.g., the clause suchthat E $\triangleright$ typeobject EqAble ... end EqAble states that parameter E must match the type defined in this clause. Matching does the following: it uses the actual parameter to instantiate the type being matched against (e.g., EqAble), using the type name in that definition as a parameter, and then checks whether the actual parameter is a derived subtype of the result. For example, if y has the type MemberObjType, then to check the call y.member[Integer, ...], Emerald would substitute Integer for EqAble within the definition of EqAble, to obtain a type with a single method function equal [Integer] $\rightarrow$ [Boolean], and would then check whether Integer was a subtype of this type. (The mechanism in PolyTOIL [BSvG95] is similar to Emerald's but the details of the matching rule are different.)

Emerald does not allow independent type definitions to be introduced to express constraints [BH95]; a new type must be defined in each matching clause. Thus, Emerald matching clauses are similar to where clauses, since the required methods are all written out explicitly. While Rapide and PolyTOIL allow independent type definitions to be used to express constraints, such types will probably never be used as normal types (e.g., there will never be objects with actual type HasEql(Integer)).

So far we have been concerned with the *ease* of expressing constraints. Another issue is semantic expressiveness. A common way to formalize type systems for object-oriented languages is to use F-bounded quantification [CCH+89]. The languages we examined are either equivalent to or more powerful than F-bounded quantification. The issue is whether one can express mutual dependencies between type parameters. For example, the type of the member routine can be expressed as an F-bounded polymorphic type:

$$\begin{array}{l} \forall\, E \subseteq \{\ \text{equal: } E \rightarrow \text{bool}\ \}. \\ \quad \forall\, C \subseteq \{\ \text{elements: void } \textit{yields}\ E\ \}. \\ \qquad C, E \rightarrow \text{bool} \end{array}$$

However, a routine with mutual dependencies, e.g., where a constraint on type parameter E mentions type parameter C and vice versa, does not have an F-bounded type, assuming the usual case where the $\forall$ construct only mentions one type at a time. For example, F-bounded quantification cannot describe the constraints in the following routine, which is used to find paths in many different kinds of graphs:

```
findpath[N, E] (s, d: N) returns (array[N])
                         signals (no_path)
    where N has edges ( ) yields (E),
                equal (N) returns (bool),
          E has source ( ) returns (N),
                dest ( ) returns (N)
```

Languages whose type systems are based on F-bounded quantification (such as Rapide) cannot express the type of findpath, while other languages, such as Theta, School, and Emerald, can. (An extension of F-bounded quantification to support mutual dependencies, e.g., by allowing multiple parameters to the $\forall$ construct, would thus have some practical value.)

4 Combining the Two Polymorphisms

A language with mechanisms for both parametric and subtype polymorphism must define how they interact. In particular, it must define the subtype relations that hold for instantiations of parameterized types. For languages such as Theta that require declared subtype relations, this translates into the question of what relationships can be declared.

```
map = type[K, V] < set[K]
      {insert_default for insert, keys for elements}
    where K has equal(K) returns (bool)
  insert (key: K, value: V)
    % Adds a mapping from key to value,
    % replacing the previous mapping, if any.
  insert_default (key: K)
    % Adds a mapping from key to a default value,
    % replacing the previous mapping, if any.
  keys ( ) yields (K)
    % Yields all the keys in the map, each exactly once,
    % in arbitrary order
  fetch (key: K) returns (V) signals (not_found)
    % Returns the value that key is mapped to. If it is
    % not mapped to anything, signals not_found.
end map
```

Figure 5: A Parameterized Subtype Example

One relationship that is clearly needed is the following: Suppose P1 and P2 are both parameterized types, with a single type parameter, and suppose P2 acts like a subtype of P1, i.e., it has the necessary methods with compatible signatures and behavior. Thus we need:

- P2[T] < P1[T] for all T

This kind of subtype relation can be declared in Theta, as shown in Figure 5. Map actually has more parameters than set, though typically parameterized types in a subtype relation have the same number of parameters. Map is a legal subtype of set because its where clause is strong enough to ensure that set[K] is a legal instantiation whenever map[K,V] is legal, and its methods' signatures conform to those of the associated set methods.

In addition to the subtype relationship just discussed, it may seem that we also want:

- P[S] < P[T] when S < T

However, this form is actually not very useful because it only works for the few types that do not use a parameter type as (a component of) an argument type of any method. For example, set has such a method, insert. Suppose S < T. Then set[S] is not a subtype of set[T] because insert(S) does not conform to insert(T), due to the contravariance rule. Such a situation occurs in types (like set) with mutable objects (because of methods that change the state of their object), and also in immutable types if they have a comparison operator such as equal. Therefore, this subtype relationship can rarely make sense.

There is a practical *semantic* reason for not allowing a subtype relationship between types such as set[S] and set[T]. For example, suppose we have a hierarchy that relates various kinds of animals, and we have declared that types elephant and rhinoceros are subtypes of type mammal but are not subtypes of one another. If we allowed set[elephant] to be a subtype of set[mammal], then it would be legal to pass a set[elephant] object to a procedure that takes a set[mammal], and in this procedure it would be legal to insert a rhinoceros object into the set, since a rhinoceros is a mammal. But the result is that our set of elephants now contains a rhinoceros! (It's interesting to note that this problem was first pointed out in 1978 in the context of access control [JL78].)

Another curious point is that sometimes the subtype relation goes in the opposite way:

- P[T] < P[S] when S < T

This relation makes sense whenever P is an "output-only" abstraction, such as an output stream. The relationship is ruled out, however, for any type with a method that returns an object of the parameter type (because of the covariance rule).

We do not permit these kinds of subtype relations in Theta, for two reasons: the relations are rarely interesting, and they are not compatible with the way we implement method calls using multiple *dispatch vectors*. Theta is a heap-based language in which objects exist in the heap and variables point to them. An object has a header that contains pointers to one or more dispatch vectors; a dispatch vector is an array of pointers to methods of the object. An object reference always points to the place in the header that points to the dispatch vector that should be used for that reference's declared type. References may need to be adjusted as part of an assignment. For example, suppose that S < T, and objects of type S contain two dispatch vectors, one used with references of declared type S, the other with references of declared type T. An assignment

```
x: T := y
```

where y is of type S results in x pointing to a different dispatch vector (a different place in the object's header) than y does. This approach allows for fast method dispatch; details are given in [Mye94, Mye95].

The implementation problem is the following: A reference to S object y from within an array[T] object would point to y's T dispatch vector, while a reference from within an array[S] object would point to y's S dispatch vector. This causes a problem if an array[S] can be passed where an array[T] is expected. It is not clear that there is a way to fix this problem without abandoning the efficient implementation! For example, copying the array[S] object to change all the pointers doesn't work because it doesn't preserve sharing (and of course it's expensive too).

5 Modules and Encapsulation

The main issue in designing a mechanism to implement abstractions is providing *encapsulation* so that local reasoning about the correctness of an implementation is sound [LG86]. The encapsulation unit could be the implementation of just one abstraction, or of many abstractions. We chose the latter, for two reasons: we need a way to implement a type and some creation routines at the same time, and we believe it is generally useful to implement several types and associated routines together.

Thus, types and routines are implemented by *modules*, and the question this raises is whether or not modules should be parameterized. Parameterizing modules (as in Modula 3 [Nel91]) could be convenient: information about parameters and constraints could be stated just once, in the module header. However, the approach has an important drawback: it does not allow a single module to contain implementations of abstractions with different parameters and different where clauses.

Therefore, modules in Theta are not parameterized. Instead, a module contains a number of implementations, some of which might be parameterized. A typical module contains a *class* that implements some type and one or more routines that create objects of that type. A module provides an encapsulation barrier: Code inside the module has access to implementation details, including the instance variables of any classes in the module, special *constructors* (one for each class) that create new objects of the class, and any local methods and routines, but code outside the module does not. The module header lists routine implementations it is exporting; routine implementations not listed are local to the module. Type implementations are exported implicitly if an exported routine returns an object of that implementation.

```
module implements hash_map_create

mapC = class [K, V] for map[K, V]
     where K has equal (K) returns (bool)
  pair = record[key: K, value: V]

  % instance variables
  default: V
  hash: proc(K) returns (int)       % the hash function
  buckets: array[array[pair]]

  % methods
  fetch (key: K) returns (V) signals (not_found)
    i: int := hash(key)
    for p: pair in buckets[i].elements( ) do
     if p.get_key( ) = key then  % call on equal method of K
        return (p.get_value( )) end
      end except when bounds: end  % no entries for key
    signal not_found
    end fetch
  ...
end mapC

hash_map_create[K,V] (h:proc(K) returns(int), default:V)
        returns (map[K,V])
      where K has equal (K) returns (bool)
  pair = record[key: K, value: V]
  return (mapC[K, V]{
           default := default,
           hash := h,
           buckets := array_new[array[array[pair]]( )})
end hash_map_create

end  % module
```

Figure 6: Example Implementation

An example of a module is given in Figure 6. The figure shows part of a class mapC that provides a simple hashed implementation of the map type; the header of the class states the type being implemented. The creation routine, hash_map_create, takes the default value and the hashing function as arguments; its behavioral specification would state that the hashing function must return a positive integer. Hash_map_create uses the special mapC constructor (in the expression mapC[K,V]{...}) to create a new object. A constructor creates an object with a slot for each instance variable of its class and a pointer to a dispatch vector that contains pointers to the object's methods (there might be several dispatch vectors, as discussed in [Mye95]); all instance variables are then initialized to the provided values. Hash_map_create uses a built-in creation routine for arrays: array_new returns a new, empty array with low bound 1. (Arrays in Theta can grow dynamically.)

In the example, the class has the same set of type parameters as the type it is implementing, but this is not required. A class may have fewer parameters than the type it implements, by requiring one or more of the type's parameters to be instantiated with a specified type. For example, a non-parameterized class could use a bit string to implement set[char]. Similarly, a class may have *more* parameters than the type it implements, or it might have more where clauses; for example, a sorted implementation of set might require an lt method in addition to the equal method.

Theta allows a class to inherit code from a single superclass. The type hierarchy is separate from the inheritance hierarchy; the subclass need not implement a subtype of the type implemented by the superclass. Details can be found in [LCD+94].

6 Implementation

This section presents an efficient implementation of parameterized abstractions for Theta. Our solution is inspired by the way parameterization was implemented for CLU [ALS78]. We assume that code is not recompiled for each actual parameter type, since this approach leads to code duplication. The scheme described here is compatible with selector-table-indexed dispatch schemes [DMSV89, Mye95, Ros88, Str87]. A more complete description of our technique can be found in [ML94].

Because parameterized code is shared by all instantiations, the key issue is how the code accesses the methods described by the where clauses. For example, consider the mapC class from Figure 6. The same code is used for all instantiations of this class; thus a mapC[string,int] object shares code with a mapC[employee,office] object. The code for the fetch method of mapC must call the appropriate equal method for the type parameter K. For example, when fetch is invoked on a mapC[string,int] object, string's equal method must be used, but if it is invoked on a mapC[employee,office] object, employee's equal method must be used.

Consider a method call x.fetch(), where x is a variable of type map[string, int]. The information about which equal method to call cannot be passed as an extra (hidden) argument to the call, because the compiler does not know x's class, and the class determines which methods are required. As discussed in Section 5, x's class might have different where clauses and even different parameters than its declared type.

We provide the needed information using the object's dispatch vector. The code for fetch calls the equal method by dispatching through a special dispatch vector entry. (We ignore the fact that objects may have several dispatch vectors, but the technique described here is easily generalized to handle that case.)

The dispatch vector actually points to *where-routines*: stubs that implement each of the declared where clauses. A where-routine encapsulates the details of calling the parameter's method. Usually, the where-routine just invokes the appropriate method of the actual type, but the Theta compiler also produces special-purpose where-routines with an inlined implementation. For example, for a mapC[int,...] object, the where-routine can compare integers directly rather than invoking the int equal method, since integers are not represented as full-fledged objects.

Figure 7 shows the necessary data structures for a mapC[string,int] object. In the figure, the dispatch vector slot labeled K.equal points to a where-routine. This slot is located at the same offset in the dispatch vector in all objects of mapC or any subclass of mapC. The where-routines are placed at negative offsets in the dispatch vector, as if they were private methods of the class [Mye95], but negative offsets are not essential.

Routines are implemented as if they were objects with a single invoke method. For routines with where clauses, the associated where-routines are appended to the dispatch vector of the routine object. When a constant routine is called (i.e., the routine can be

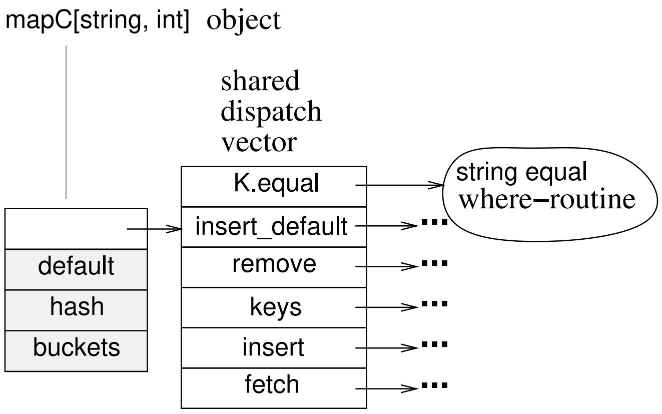

Figure 7: Object Layout

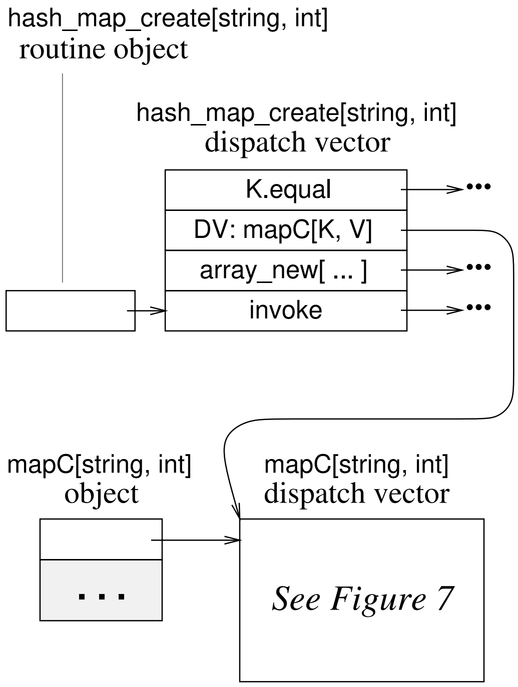

Figure 8: Routine Object Layout

identified at link time), the method call to invoke is optimized away and the call is made directly. In any case, the where-routines are accessible through the routine's dispatch vector, since the routine object is implicitly passed as the self argument to its own code. (Routine objects are also useful for implementing currying, which is supported by Theta. Currying produces a routine object with fields that contain the curried arguments.)

Routine objects resulting from instantiations are usually constructed at link time, using a template (produced by the compiler) that describes the necessary components of the routine dispatch vector. A parameterized routine often contains an instantiation of some other parameterized routine within it. For example, hash_map_create uses an instantiation of the array creation routine, array_new. This instantiation could be done each time hash_map_create runs, but that would be unnecessarily costly. Instead, contained instantiations are also performed at link time, as part of doing the instantiation of the containing code, and the dispatch vector produced for the containing code points to the resulting objects. Thus the linker arranges that the dispatch vector of every hash_map_create routine object contains a pointer to the corresponding array_new object, which is used for the call to array_new.

When an object is created, an instantiation-specific dispatch vector containing the needed where-routines and routine objects must be available. For example, the hash_map_create routine creates mapC objects using a mapC class constructor. This code must put an appropriate dispatch vector pointer into the newly-created object. This dispatch vector pointer is also stored in the dispatch vector of the calling context.

Figure 8 shows the necessary data structures for the routine object hash_map_create[string,int]. As just discussed, the dispatch vector contains a pointer to the array_new routine object that hash_map_create invokes to create a new array, and a pointer (DV) to the dispatch vector for mapC[string, int] that hash_map_create uses to build the new mapC object.

In summary, whenever a piece of parameterized code runs in the system, there is an associated dispatch vector that contains information for the particular instantiation of the code that is running. For method code, the associated dispatch vector is the dispatch vector of the method's object; for routine code, the associated dispatch vector is the dispatch vector of the routine object. In addition to the usual entries for methods, a dispatch vector contains three kinds of entries:

1. A call to a method described in a where clause requires a pointer to a stub routine that handles the appropriate method invocation, e.g., K.equal in Figure 7.

2. Use of a parameterized object constructor requires a pointer to the dispatch vector that is installed in the object, e.g, DV in Figure 8.

3. A call to a routine instantiation requires a pointer to the object for the instantiation, e.g., array_new in Figure 8.

The format of the dispatch vector is determined by inspecting the actual code of the parameterized implementation. Since calling code does not depend on the layout of (this part of) the dispatch vector, this inspection does not prevent separate compilation.

Dispatch vector entries can be filled in at link time, since the identity of all classes and routines involved is known at that point. In the Theta implementation, the linker performs instantiations lazily, generating only those dispatch vectors that are needed by existing code. Since some dispatch vectors contain pointers to other dispatch vectors or to procedure objects that contain dispatch vectors, the instantiation process is recursive. In certain pathological implementations, the instantiation process cannot complete, because infinitely many instantiations must be generated [ALS78]; linking fails in this case. The problem could be solved by deferring some instantiations until run time, but recursive instantiations are not a problem in practice.

The technique described here has low cost in terms of both space and time. It is fast because most work is done at link time. A call to a where-routine is roughly twice as expensive as an ordinary method call. All objects belonging to the same instantiation of a parameterized implementation share the same dispatch vector(s). This implementation technique also supports specialized implementations of parameterized types, such as the special implementation of set[char] that uses bitstrings.

7 Conclusions

In this paper we have argued that full support for the construction of generic code requires both subtype polymorphism and parametric polymorphism. Our main contribution is to bring together in one place a discussion of all of the language design issues involved in providing both kinds of polymorphism. In addition, using a new programming language, Theta, as an example, we have presented a coherent set of design decisions.

We showed that support for modular programming and separate compilation requires a mechanism for specifying constraints on the type parameters of a parameterized abstraction. We explained why subtypes cannot be used to capture constraints, and we presented a simple alternative, where clauses. We compared where clauses with constraint mechanisms in other languages such as Emerald and Rapide. Where clauses are both powerful and easy to use.

We discussed the advantages of having declared subtyping as opposed to derived subtyping: declared subtyping produces a finer-grained type system, allowing the type checker to catch more errors, and it allows for method renaming. We also discussed the relationship between subtype and parametric polymorphism by analyzing the subtype relations that exist among instantiations of parameterized types. We argued that it is desirable to support only relations of the form P2[T] < P1[T] for all T. We ruled out other relations, such as P[S] < P[T] when S < T, since they are rarely valid and the limitation allows for a very efficient implementation of method dispatch. We also described a way of implementing parameterized routines and types that is efficient in both space and time.

Acknowledgments

The authors gratefully acknowledge the help given them by Andrew Black, Luca Cardelli, Norm Hutchinson, Roberto Ierusalimschy, John Mitchell, Raymie Stata, members of the Theta design group, and the referees.

References

[ALS78] Russell Atkinson, Barbara Liskov, and Robert Scheifler. Aspects of implementing CLU. In *Proceedings of the ACM 1978 Annual Conference*, October 1978.

[BH95] Andrew Black and Norman Hutchinson, August 1995. Personal communication.

[BHJ+87] 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.

[BSvG95] Kim Bruce, Angela Schuett, and Robert van Gent. Polytoil: A type-safe polymorphic object-oriented language. In *ECOOP'95*, 1995.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In *Semantics of Data Types, LNCS 173*, pages 51–68. Springer-Verlag, 1984.

[CCH+89] 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.

[Coo89] William R. Cook. A proposal for making Eiffel type-safe. In *ECOOP'89*, pages 52–72, October 1989.

[DMSV89] R. Dixon, T. McKee, P. Schweitzer, and M. Vaughan. A fast method dispatcher for compiled languages with multiple inheritance. In *OOPSLA '89 Conference Proceedings*, pages 211–214, New Orleans, LA, October 1989. Published as *SIGPLAN Notices 24(10)*, October, 1989.

[ES90] Margaret A. Ellis and Bjarne Stroustrup. *The Annotated C++ Reference Manual*. Addison-Wesley, 1990.

[JL78] Anita K. Jones and Barbara Liskov. A language extension for expressing constraints on data access. *CACM*, 21(5):358–367, May 1978.

[KLMM94] Dinesh Katiyar, David Luckham, John Mitchell, and Sigurd Melda. Polymorphism and subtyping in interfaces. *ACM SIGPLAN Notices*, 29(9):22–34, August 1994.

[LCD+94] Barbara Liskov, Dorothy Curtis, Mark Day, Sanjay Ghemawhat, Robert Gruber, Paul Johnson, and Andrew C. Myers. *Theta Reference Manual*. Programming Methodology Group Memo 88, MIT Laboratory for Computer Science, Cambridge, MA, February 1994. Also available at http://www.pmg.lcs.mit.edu/papers/thetaref/.

[LG86] Barbara Liskov and John Guttag. *Abstraction and Specification in Program Development*. Cambridge MA: MIT Press, New York: McGraw Hill, 1986.

[LSAS77] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert. Abstraction mechanisms in CLU. *CACM*, 20(8):564–576, August 1977.

[LW94] Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM TOPLAS*, 16(6):1811–1841, November 1994.

[Mey86] Bertrand Meyer. Genericity versus inheritance. In *Proceedings of OOPSLA '86*, September 1986.

[ML94] Andrew C. Myers and Barbara Liskov. *Efficient Implementation of Parameterized Types in an Object-Oriented Language*. Programming Methodology Group Memo 91, MIT Lab for Computer Science, July 1994. Also available at ftp://ftp.pmg.lcs.mit.edu/pub/thor/param-impl.ps.gz.

[Mye94] Andrew C. Myers. *Fast Object Operations in a Persistent Programming System*. Technical Report MIT/LCS/TR-599, MIT Laboratory for Computer Science, Cambridge, MA, January 1994. Master's thesis.

[Mye95] Andrew.C. Myers. Bidirectional object layout for separate compilation. In *OOPSLA '95 Conference Proceedings*, Austin, TX, October 1995.

[Nel91] Greg Nelson, editor. *Systems Programming with Modula-3*. Prentice-Hall, 1991.

[RIR93] Noemi Rodrigues, Roberto Ierusalimschy, and José L. Rangel. Types in school. *ACM SIGPLAN Notices*, 28(8):81–89, August 1993.

[Ros88] John R. Rose. Fast dispatch mechanisms for stock hardware. In *OOPSLA '88 Conference Proceedings*, pages 27–35, San Diego, CA, October 1988. Published as *SIGPLAN Notices 23(11)*, November, 1988.

[SCW85] 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.

[Str87] Bjarne Stroustrup. Multiple inheritance for C++. In *Proceedings of the Spring '87 European Unix Systems Users's Group Conference*, Helsinki, May 1987.