Massachusetts Institute of Technology
Laboratory for Computer Science

Thor Note 9
July 13, 1994

Efficient Implementation of Parameterized Types Despite Subtyping

Andrew Myers   Barbara Liskov

This note explains how to efficiently implement parameterized types in Theta. Theta has subtyping, multiple implementations, and run-time type discrimination, in addition to sophisticated parameterized-type features such as optional methods and individually parameterized methods. The interaction of these features makes the implementation fairly complex; however, all these features can be supported with minimal runtime overhead.

1 Subtyping without Parameterization

First, let's look at how ordinary types are implemented in Theta. Each object contains a pointer to a *dispatch vector* containing pointers to the code implementing the methods. For example, an object of type list, representing a heterogeneous singly-linked list

```
list = type
  first() returns(any)
  rest() returns(list)
end
```

will have the layout shown in Figure 1.

1.1 Method Invocation

For a method invocation, the compiler generates code to index into the dispatch vector, fetch the appropriate code pointer, and jump to the code.

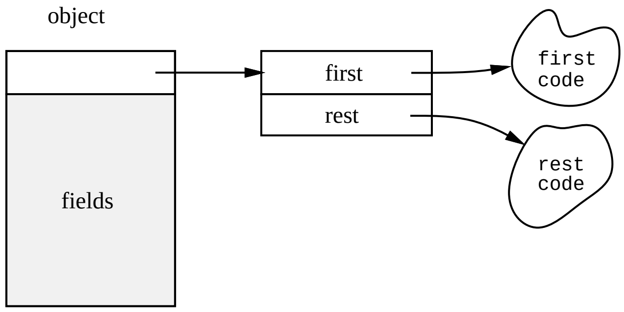

Figure 1: Object layout

This object layout means that we can invoke an operation on an object of type list (or any other object type) without knowing the actual implementation of the object.

Note that the dispatch vector can be shared by all objects of a particular implementation, since it is the same for all of them.

1.2 Type Extensions

In addition to simple method invocations, Theta also allows invocation of *type extension operators*. Operations that create new objects are typically defined in type extensions because they do not require an existing object of the type. A type extension is similar to an ordinary type specification. For example, listExt might define list creators (cons might be considered to be a method rather than a creator, but this is a side issue):

```
listExt = extension list
  cons(head: any, tail: list) returns(list)
  empty() returns(list)
end listExt
```

We implement type extensions by treating them as ordinary objects that belong to a type whose specification is similar to the type extension:

```
listExtType = type
  cons(head: any, tail: list) returns(list)
  empty() returns(list)
end listExtType
```

Thus, a call on an extension operation like listExt.empty() is understood in implementation terms to be an invocation of the method empty on an *extension* *object* of type listExtType. The extension object has an dispatch vector whose layout is determined by its type, just as do ordinary objects. The actual selection of a particular extension object used in an extension operator call is performed by the linker.

Note that extension objects do not have any state in the current implementation; however, if class variables were added to Theta, they would be implemented as fields of the extension object. Other than being stateless, they act just as normal objects do. Extension objects are automatically created at the same time that their corresponding class is linked.

1.3 Runtime Type Information

In Theta, the typecase statement may be used to determine the actual type of an object. For example, in Theta the type any is the supertype of all object types. The following code determines whether a value of apparent type any is actually a list and returns its first element if so:

```
x: any
...
typecase x
   when list(l): return(l.first())
end
```

To support typecase, each object or its dispatch vector must contain some runtime type information. This information is used to determine the actual type of the object, and thus the proper arm of the typecase to execute.

2 Parameterization without Subtyping

Let's look at how parameterized types are implemented in CLU [1], with some simplifications that do not substantially affect this discussion. Consider the CLU type ordered_map.

```
ordered_map = cluster[K,V]
        where K has equal(x,y: K) returns(bool),
                    lt(x,y: K) returns(bool)
  size(m:ordered_map[K,V]) returns(int)
  fetch(m:ordered_map[K,V], key:K) returns(V) signals(not_found)
  store(m:ordered_map[K,V], key:K, value:V)
  % there would also be some creators!
end ordered_map
```

In CLU, a call to an type operation is implemented by jumping directly to the operation's code, since each type has only a single implementation. However, different *instantiations* of ordered_map, such as ordered_map[int, int] or ordered_map[string, array[int]], differ in the procedures that are used to satisfy the

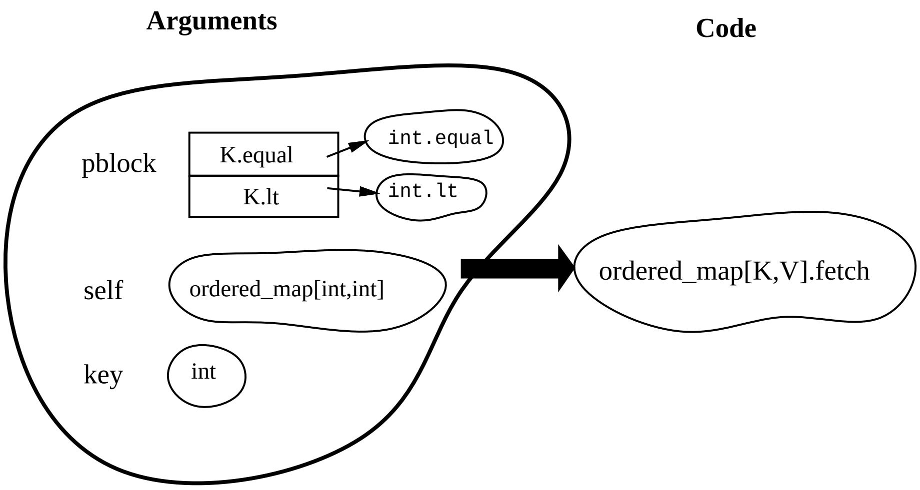

Figure 2: CLU calling sequence

where clauses. For example, if the code for the fetch operation of ordered_map contains the statement

```
k1,k2: K
if K$equal(k1,k2) then return (v1) end
```

the call to K$equal results in a call to a different procedure for each choice of K.

2.1 P-blocks

To avoid recompiling this statement for each distinct K, the CLU implementation passes an extra argument, called the *entry block* or *p-block* (for "parameter block") to the procedure that implements fetch. The *p-block* is a vector that contains one entry for each of the parameter routines. This calling sequence is illustrated in Figure 2 for the type ordered_map[int, int].

However, more information than just the parameter routines is needed in the p-block. Consider what happens if there is a call to an array[K] operation inside the ordered_map implementation:

```
keys: array[K] := ...
l:int := array[K]$length(keys)
```

where the specification of array is

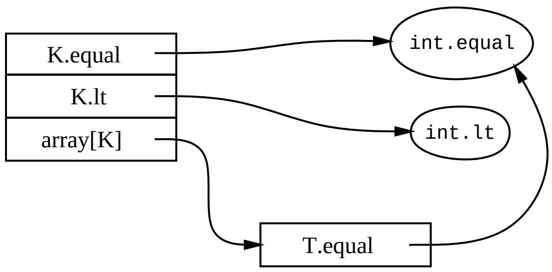

Figure 3: The p-blocks for ordered_map[int,int] and array[int]

```
array = cluster[T] where T has equal (T) returns (bool)
  length(a: array[T]) returns(int)
  ...
end array
```

The call on array[K]$length() requires a p-block for array[K] [1]. In CLU, a pointer to the array[K] p-block is placed in the p-block for ordered_map[K,V] (for all V), so that the ordered_map implementation can call array operations. Therefore, the ordered_map[int,int] p-block contains, in addition to pointers to int operations, an auxiliary p-block pointer for each instantiation used in the ordered_map implementation. This p-block is shown in Figure 3.

Thus, a p-block contains both pointers to the parameter routines (or *p-routines*) and some auxiliary pointers. Importantly, the structure of the p-routines part can be understood from just the specification of the parameterized type, whereas the layout and contents of the the auxiliary information can be determined only by looking at the *implementation* of the type. In fact, CLU p-blocks only contain p-routine information if the implementation actually uses the p-routines.

Since the content of the p-block for a type depends on the particular implementation chosen for the type, the contents of the p-block are filled in at link time in CLU. A pointer to the p-block is stored with the code of the caller so that the p-block is available at the point of the call. The compiler indicates to the linker where to store the pointer, and the linker fills in this information when it creates the p-block at link time.

2.2 Avoiding Recursion

In fact, infinite chains of p-blocks may be theoretically required to support some procedures. For example, consider the following legal CLU example from [1],

[1] Actually, CLU has a separate p-block for each operation of array[K], but the distinction is not important.

which contains a recursive instantiation:

```
f = proc[T: type](n: int) returns(any)
   if n = 0 then
      return (array[T]$new())
   else
      return (f[array[T]](n - 1))
   end
end f
```

Since f calls itself with the parameter array[T], a call to f with some positive n requires n distinct instantiations, for f[T], f[array[T]], f[array[array[T]] ...

This code example causes the CLU linker to go into an infinite loop, producing p-blocks. Avoiding this fate would be desirable in Theta.

2.3 Stand-alone Routines

Although the above discussion has considered only parameterized types, the same approach is used when calling instantiations of parameterized stand-alone routines (e.g., a call on sort[int]). Again there is a p-block as an extra argument, and it contains both p-routines and any auxiliary information needed by the implementation of the called procedure. Also, as was the case with type instantiations, the p-block is filled in at link time, and the code of the caller contains a pointer to the p-block.

2.4 Unoptimized CLU

The CLU implementation is actually an optimization of a more straightforward scheme in which the caller passes in a p-block containing only p-routines and the callee always dynamically constructs the p-blocks for its parameterized calls, using the information in its p-block. Thus, within the ordered_map implementation the p-block for a call of a array operation would be constructed by copying the pointer to the equal routine of ordered_map's p-block into the slot for the equal routine of array's p-block. This scheme does not require the linker to know about implementation details of the callee when it links the caller, but it has the disadvantage that p-blocks are constructed at runtime. The CLU scheme avoids most runtime overhead by using a dynamic linker to build p-blocks that are then reused.

3 Parameterization and Subtyping

Both parameterization and subtyping are present in Theta, and we can support them by integrating the solutions described in the previous sections. Our goal is to support parameterization as efficiently in Theta as in CLU.

Since the auxiliary information in the p-block depends on knowing the implementation, the CLU approach of passing p-blocks does not work in a system with subtyping and multiple implementations. In particular, the auxiliary information cannot in general be known at link time. For example, consider an ordered_map type in Theta with the following interface:

```
ordered_map = type[K,V]
        where K has equal(x: K) returns(bool),
                 lt(x: K) returns(bool)
  size() returns(int)
  fetch(k:K) returns(V) signals(not_found)
  store(k:K, v:V)
end ordered_map
```

When an ordered map object is used, e.g.,

```
x.store(3,7)      % where x is an ordered_map[int, int]
```

we don't know its implementation and therefore there is no way that the linker can provide the auxiliary information. However, this information was known at the point where x was created, since at that point we used the extension object of a particular implementation, and so the information can be stored in x's dispatch vector. P-routine information *could* be passed via an external p-block, but the runtime system will be simplified and programs will perform better if we do not require an extra argument for method calls. Instead, the entire p-block — p-routines and auxiliary information — is placed directly in the dispatch vector.

Consider our earlier example of an ordered_map[int,int] implementation that uses array[K] methods. The ordered map object is structured as shown in Figure 4. Note that the code implementing the ordinary methods has been omitted for simplicity.

As this figure shows, there are now three categories of information in the dispatch vector. The "methods" part is exactly as before. Let's examine the other two parts.

4 P-routines

As in CLU, the p-routines point directly to the associated routines. Often these routines just call appropriate methods. In this case the slots point to small stub procedures or iterators (created by the Theta implementation) that perform the appropriate method call. For example, in the p-routines for ordered_map[foo,bar], the K.equal slot points to a procedure of the form:

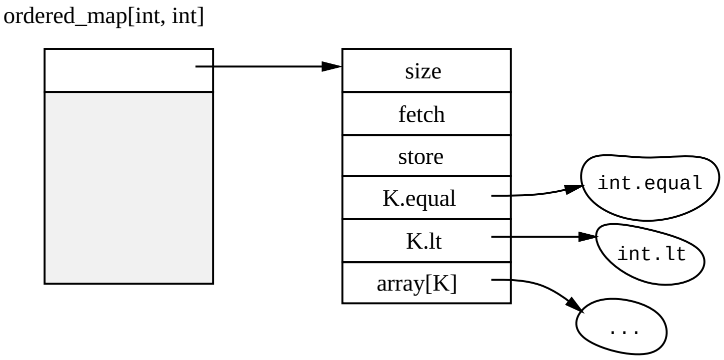

Figure 4: Layout of an ordered_map[int,int] object

```
equal_stub (x,foo: K) returns(bool)
   return(x.equal(foo))
   end equal_stub
```

If the instantiation replaces the equal operation by using a for clause, as in ordered_map[foo{op a for equal}, bar], the p-routine slot for K.equal points directly to the procedure a. If the instantiation replaces one operation for another, as in ordered_map[foo{similar for equal}, bar], the stub procedure simply calls the substituted method:

```
equal_stub(x,foo: K) returns(bool)
   return(x.similar(K))
   end equal_stub
```

5 Auxiliary Information

Auxiliary information is required in the dispatch vector because it includes runtime type information to support the typecase operation. For example, in the following typecase example, two instantiations of ordered_map must be distinguished. An implementation based solely on passing p-routines (such as the unoptimized CLU scheme mentioned earlier) cannot install the proper runtime type information in every object.

```
foo[T](y: T)
   x: any
   ...
   typecase x
```

```
        when ordered_map[int, char](map):
            return(map.fetch(1))
        when ordered_map[T, char](map):
            return(map.fetch(y))
    end
end foo
```

The runtime type information may be placed in the dispatch vector or in the object; in either case, it is auxiliary information that cannot be supplied by the p-routines.

Since objects now contain their own p-blocks, there is no need to store auxiliary p-block pointers for ordinary method invocations, e.g., a call of an slist method within an ordered_map method. Auxiliary information is needed to support calls to a parameterized abstraction. This can occur in three different kinds of invocations: a parameterized type extension, a parameterized stand-alone routine, or a parameterized method. Here we discuss the first two cases; parameterized methods are discussed in Section 11.

As described earlier, type extensions are represented by objects whose methods are described by the type extension. Consider the parameterized extension arrayExt:

```
arrayExt = extension array[T] where T has equal(T) returns (bool)
   create() returns (array[T])
end arrayExt
```

If the code for ordered_map contains a call to this type extension, e.g., arrayExt[K].create(), each possible K corresponds to a different extension object. For each implementation that uses the parameterized extension arrayExt, the dispatch vector of that implementation contains a pointer to the appropriate extension object instantiation.

Armed with this information, we can fill in more details of the diagram of Figure 4, as shown in Figure 5. In this figure, a pointer to the appropriate extension object is found in the auxiliary information part of the dispatch vector.

Calls to parameterized stand-alone routines are handled as in CLU, i.e., the p-block is passed as an extra argument. The p-block is created by the linker, since we know at link time what implementation of the routine is being called. The pointer to the p-block is stored in the calling code in the case where this is a stand-alone routine; if the call occurs in a method or an extension operation, the pointer to the p-block is stored in the dispatch vector of the method or extension operation's object.

6 Makers

The above strategy depended on having p-blocks stored in dispatch vectors. We now consider how this information gets into the dispatch vector in the first

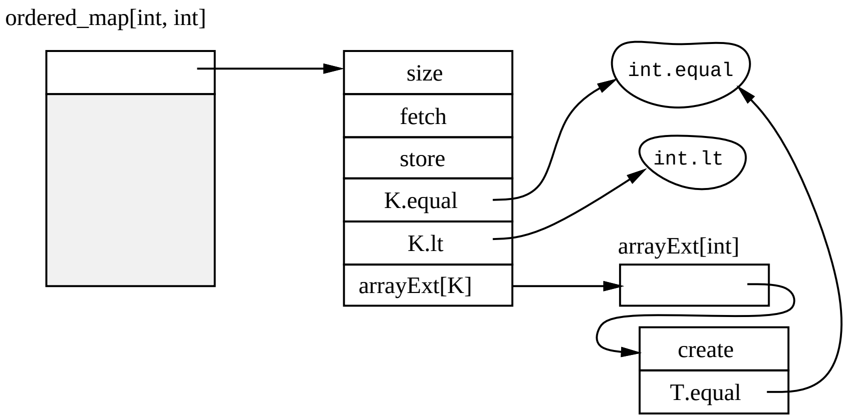

Figure 5: Layout of ordered_map[int,int] and arrayExt[int] objects

place. Clearly this must happen when the object is created, i.e., when the maker that creates it runs.

A maker is responsible for initializing the object and installing its dispatch vector pointer(s). For a maker that is a method, installing the correct pointer is easy, because it can be copied from self. An op maker does not receive a self object, so that the dispatch pointer (or some equivalent information, such as the metaobject representing the parameterized class being created) must be found somewhere else. A good place to store this information is in the dispatch vector of the extension object. If the type is parameterized, there will be a different extension object for each instantiation, and an instantiation's extension object will contain a pointer to the proper dispatch vector for that instantiation's objects. Thus, the dispatch vector for arrayExt[int] contains a pointer to a dispatch vector for an implementation of array[int] in addition to the slots that are depicted in Figure 5. The complete layout is shown in Figure 6.

7 Runtime Type Discrimination

In a system without parameterization, runtime type discrimination is fairly straightforward. Each object contains type information that is compared against the types mentioned in the typecase statement to determine which arm of the typecase to perform. Exactly what type information is stored depends on the implementation of typecase, and is not important for this discussion.

In the presence of parameterization, objects may have parameterized types, and the types mentioned in the typecase may be parameterized as well. Consider the earlier example:

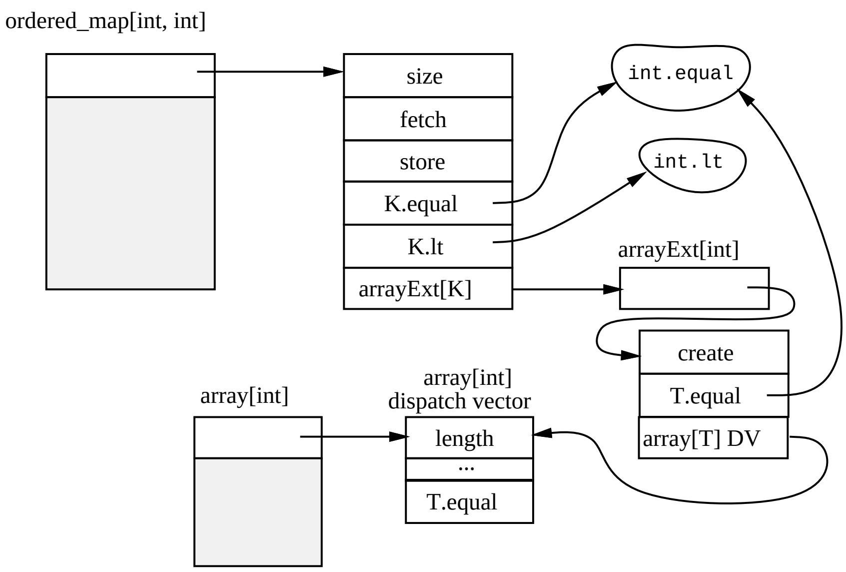

Figure 6: Complete layout of ordered_map[int,int] and arrayExt[int] objects

```
foo[T](y: T) where T has equal(T) returns (bool)
   x: any
   ...
   typecase x
      when ordered_map[int, char](map):
         return(map.fetch(1))
      when ordered_map[T, char](map):
         return(map.fetch(y))
   end
end foo
```

Since the type T is a parameter of this code, the type represented by ordered_map[T, char] is not fixed. For example, T may even be int, so that both arms describe the same type. This does not create any new semantic problems; because of subtyping, multiple arms of the typecase statement may match the run-time type of an object. In this case, the first matching arm is executed.

To support determination of the object's type information, the run-time type information for the object is placed in the object's dispatch vector. This works because different instantiations of the object's class have different dispatch vectors.

Arms of the typecase statement that mention partially-bound instantiations (such as ordered_map[T, char], above) are supported by storing type information into the dispatch vector for the object that owns the code. This information is stored in the auxiliary portion of the dispatch vector, just as are type extensions and dispatch vectors for makers.

8 Dispatch Vector Layout

The dispatch vector now contains several kinds of information. The location of this information must be known even in the presence of subtyping and inheritance. The general rule that governs dispatch vector layout is that a type's dispatch vector must be compatible with that of its supertypes; similarly, a class's dispatch vector must be compatible with that of its superclass.[2]

In Theta without parameterization, the dispatch vector for type T starts with embedded dispatch vectors for supertypes of T. This layout still holds, but the p-routines for T are placed after the methods for T, and before the methods of subtypes of T. In other words, the param ops are treated as if they were methods of the parameterized type in which they were declared.

Since the auxiliary information is associated with the particular operations performed in an implementation, auxiliary information is stored in the implementation-specific part of the dispatch vector, which contains just the private (class-only) methods in unparameterized classes.

To see how these rules behave in practice, consider the parameterized types slist and dlist, where dlist is a doubly-linked list type that also supports all the operations of slist.

```
slist= type[T] where T has equal(T) returns (bool)
  first() returns (T)
  rest() returns (slist[T])
  equal(s: slist[T]) returns (bool)
end slist


dlist= type[T] supers slist[T]
  next() returns (dlist[T])
  prev() returns (dlist[T])
end dlist
```

The corresponding dispatch vectors are shown in Figure 7. Note that if dlist added any new where clauses, the corresponding param ops would appear below the dlist methods in the dispatch vector. Any implementation of slist or dlist would append its auxiliary information to the end of the appropriate dispatch vector.

[2] A complete discussion of the layout of our dispatch vectors can be found in [2].

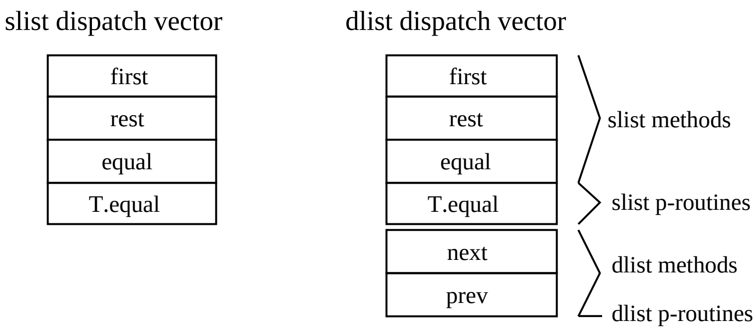

Figure 7: Dispatch vectors for slist and its subtype dlist

Actually all this discussion of dispatch vectors is a simplification, since an object may have several dispatch vectors in a system with multiple supertypes. However, even in such a system, the dispatch vector is a series of layers, where each layer contains the dispatch information for some class or type. The rules above can be applied straightforwardly to such a dispatch vector.

9 Avoiding Recursion

As we saw in Section 2.2, infinite chains of p-blocks may be required by CLU code that contains recursive instantiations. The same effect can occur in Theta, where an operation of a type extension may cause an invocation on the same parameterized type extension, but with a derived type parameter. Ordinary method invocations cannot cause recursive instantiations, because the method's caller is not responsible for providing the p-block.

The recursion is broken by inserting an incomplete data structure that is dynamically linked when used. In Theta, the incomplete data structure is the extension object; when created, a new extension object instantiation has incomplete methods; the method pointers in the dispatch vector point to procedures that will dynamically instantiate the extension object, then redispatch the extension call to the actual implementation.

This implementation technique is illustrated in Figure 8 for the following piece of code:

```
ext = extension[T] foo[T] where T has equal(T) returns (bool)
   f(n: int) returns (array[T])
end ext

...
```

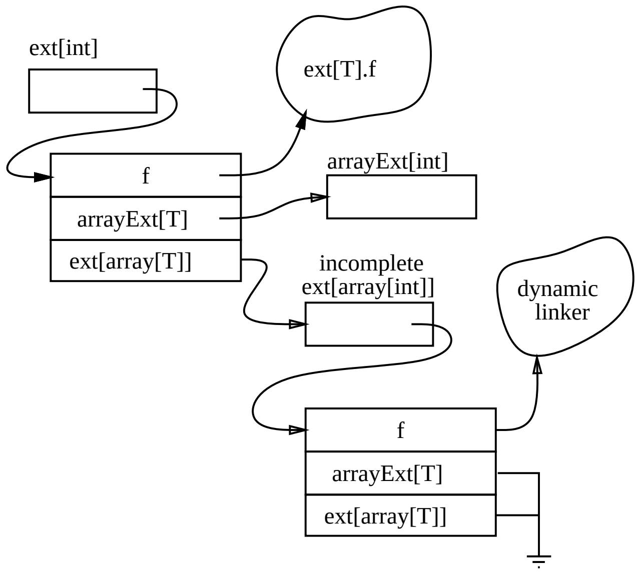

Figure 8: An incomplete type extension object

```
f(n: int) returns(array[T])
    if n = 0 then
        return arrayExt[T].new()
    else
        return ext[array[T]].f(n - 1).fetch(0)
    end
end f
```

If a call is actually performed on the incomplete ext[array[int]] object, the instantiation machinery will fill in the empty auxiliary information (signified in the figure by a connection to ground) and repair the method pointers in the dispatch vector, then call the actual f method. The auxiliary information for ext[array[T]] will contain a pointer to an newly-generated incomplete extension object for ext[array[array[int]]].

This instantiation machinery operates by reading dynamic type information located in the extension object (not shown in the illustration) in order to determine what instantiation is performed.

10 Optional Methods

Theta permits the declaration of *optional methods*: methods that exist only if additional where clauses are satisfied. For example, slist might make the equal method optional:

```
slist= type[T]
  first() returns (T)
  rest() returns (T)
  equal(s: slist[T]) returns (bool)
    where T has equal(T) returns (bool)
end slist
```

If its where clause is not satisfied, the optional method cannot be called: If foo has no equal method, calling slist[foo].equal is a static type error.

Optional methods are implemented exactly like normal methods: they take up a spot in the dispatch vector, even when the object does not support them, since the layout of the dispatch vector must be the same for all choices of the parameter type.

Similarly, the p-routines corresponding to the where clauses also take up space in the dispatch vector with the other p-routines. The dispatch vector for the new version of slist will look just as it did in Figure 7, but the last two slots will not always be occupied.

Whether or not an optional slot in the dispatch vector is filled in depends on the parameter type. For an object that does not support an optional method, the contents of the dispatch vector slots for that method are irrelevant, since no access to that information will be performed.

As we noted earlier, makers are implemented by fetching the dispatch header from the auxiliary information in the dispatch vector of the maker's object. Since all the dispatch headers are precomputed for each specific class instantiation in use, there is no extra runtime cost for filling in the method or p-routine slots of an optional method. In fact, optional methods have no associated performance penalty.

11 Parameterized Methods

Theta also permits individual methods to be parameterized. For example, we might want to compare slists even when the parameter types are different. This can be accomplished by parameterizing the equal method, giving it the signature:

```
equal[U](s: slist[U]) returns (bool)
  where U has equal(T) returns (bool)
```

This method allows us to compare an slist[t] and an slist[s], even though the types slist[t] and slist[s] have no relationship in the type hierarchy. Such a comparison is often useful when s and t have a subtype relationship.

The actual value of the parameter U is only known at the point of a call to equal, e.g., x.equal[foo](y). Therefore, the necessary information about U cannot be stored in the dispatch vector of the called object (e.g., in the dispatch vector of x).

Now consider what happens when we use U as a parameter within the slist implementation of equal. For example, the implementation of equal might contain:

```
...
a: array[U] := arrayExt[U].create()
```

An extension object is needed to support the arrayExt call. However, as noted above, the correct extension object cannot be stored in the dispatch vector of self, because it depends on U. Also, the correct extension object cannot be passed in by the caller of the parameterized method (e.g., the caller of slist's equal method), because the use of arrayExt[U] is internal to a particular implementation of slist[T], about which an external caller cannot be aware. Therefore, a combination of these strategies is required.

Our solution is as follows. The dispatch vector for an object with a parameterized method has a slot that points to a table for the method, called the *parameterized method table*. (The table is needed only for parameterized methods that have where clauses constraining their parameter.) The table maps from keys that represent unique instantiations of the parameterized method to the p-block for that instantiation. A call to a parameterized method (with a where clause) has a vector containing the p-routines as an extra argument. The uid of this vector is used as a key in the table. If there is no p-block associated with that key, one is created, using the p-routines from the argument and auxiliary information from the object's dispatch vector. The p-block in the table is used to run the call.

This process is depicted in Figure 9. The figure shows the structure of the three arguments to x.equal[U](s) (x, p, and s) in the case where x is an slist[int], p is the vector of p-routines, and s is an slist[float]. The hash-table lookup is used to find the correct version of U.equal; in this particular example, the stub code located would look something like (assuming that floats know how to compare themselves to ints):

```
equal_stub(x: float, y: int)
   return x.equal(y)
end equal_stub
```

Thus, when a parameterized method call is made, the vector of p-routines is passed as an additional argument. This leads to the question of how the calling

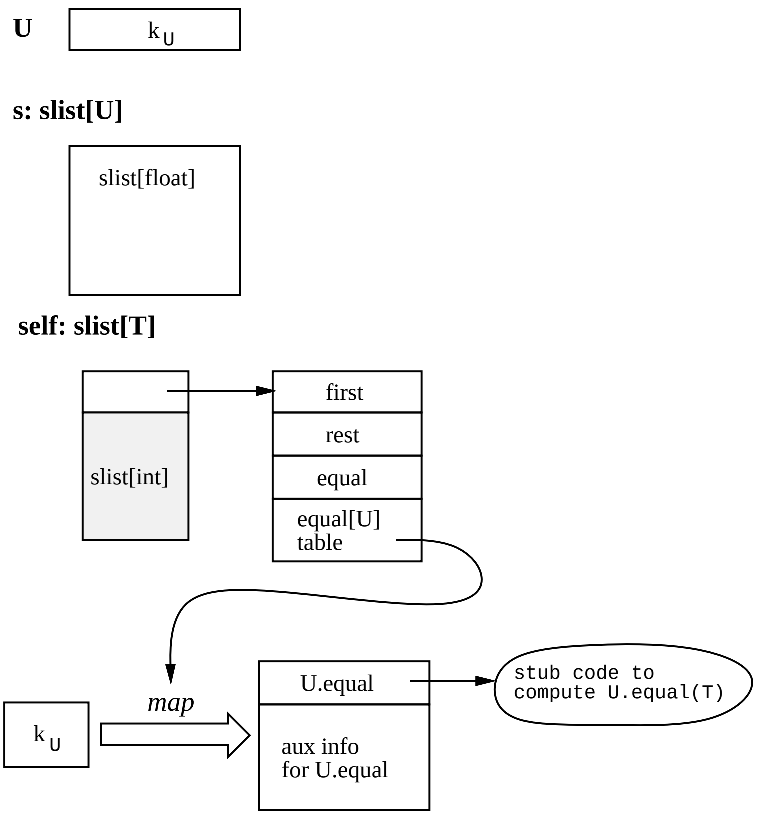

Figure 9: The arguments to slist[int].equal[float]

code gets hold of this vector. We solve this problem in the usual way: a pointer to the vector is stored in the (auxiliary information in the) object's dispatch vector, or in the hash table for a parameterized method.

Note that the where clauses of the parameterized method can include additional constraints on the parameter(s) of the type, if any. For example, ordered_map might have an equal method:

```
equal [U: type] (y: ordered_map[K,V])
   where U has equal(K) returns(bool), V has equal(V) returns (bool)
```

If these where clauses mention the parameter of the method (e.g. U), their p-routines must be placed in the parameterized method table. For simplicity, it is probably best to place *all* the information associated with the parameterized method into the parameterized method table for that method.

Because the parameterized method p-block is created dynamically as needed, the problem of recursive instantiation (encountered in extension objects) does not occur for parameterized methods, since the hash table itself serves as a mechanism for breaking the recursion.

The runtime overhead of parameterized methods is not large: it requires an extra argument and a hashed lookup in the case where the p-block already exists in the table. This overhead increases the base method-call overhead by roughly a factor of 4.

12 Acknowledgements

Our thanks to Dorothy Curtis and Paul Johnson for their helpful suggestions and their corrections of the more egregiously incorrect statements about CLU.

References

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

[2] 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.