MetaAccessClass newName: #AccessClass
superclass: Classstalk
instanceVariableNames: ""
category: 'Metaclass-Library'!

\AccessClass methodsFor: 'access generation'
makeIVAccessOn: ivNameArray
ivNameArray isNil ifFalse:
  [ivNameArray do: [:ivString |
    self compile: ivString , \" ^ \" withCRs , ivString
    classified: #accessing;
    compile: ivString , \" : \" withCRs , ivString
    classified: #accessing]]]]]]

7 Multiple Inheritance
We described examples from the library of metaclasses. The programmer may combine them by using instantiation and inheritance. In non trivial cases, simple inheritance may be not enough. Therefore we introduce multiple inheritance in Classstalk, while reusing most of the standard Smalltalk-80 extension for multiple inheritance.

7.1 MI In Smalltalk-80
The strategy proposed in [Ingalls&Borning82] is to keep the single inheritance scheme working. In case of multiple inheritance the first superclass continues to be the standard superclass, while others are stored in the metaclass of the class. These remaining superclasses are referenced by the new instance variable otherSupers, which is introduced by the kernel class MetaclassForMultipleInheritance:

Metaclass (thisClass)
MetaclassWithMultipleInheritance (otherSupers)

When creating a class with multiple superclasses, the methods which cannot be reached by the standard single inheritance lookup are recompiled into the method dictionary of the new class. If several methods with the same selector may be reached, conflicting inherited methods are automatically generated. To solve the problem, the conflicts need to be resolved by the programmer.

7.2 MI In Classstalk
When modeling multiple inheritance in Classstalk we define the instance variable otherSupers directly at the class level (and not at the metaclass level). Consequently we introduce the metaclass MIClass to define this new instance variable. As with metaclasses TypedClass and AccessClass, to extend the creation method we have to introduce a metaclass, namely MetaMIClass:

Classstalk (name category)<newName:superclass:...category:>
MIClass ()<newName:superclasses:instanceVariableNames:category:>

The method to create classes with multiple superclasses is named
newName:superclasses:instanceVariableNames:category: Its syntax and implementation are similar to those of the standard Classstalk method
newName:superclass:instanceVariableNames:category:.

8 A Developed Example: Typed Stacks
To emphasize the Classstalk methodology we develop the parameterized stacks example. Our goal is to define stacks whose parameter of the push method is typechecked. To make the demonstration easier, and to show how we may reuse standard libraries, we suppose that a class Stack has been previously defined, e.g. as a subclass of primitive class Array extended with an index. Note that Stack can be either a Classstalk class either a Smalltalk-80 class.

The class architecture we want to discuss is summarized by the following figure and steps:

```
AbstractClass
      |
      v
MIClass
      |
      v
TypedClass

Stack

AbstractTypedClass

TypedStack

IntegerStack

StringStack

IntegerStack("12")

StringStack("ok")
```

- to express the different types of stacks (IntegerStack, StringStack...), each type of stack is defined as a parameterized class (i.e. an instance of TypedClass),
- to express the common behavior (and structure) of typed stacks, we introduce the abstract class TypedStack,
- to maintain consistency between TypedStack and its subclasses (IntegerStack, StringStack...), TypedStack must be also parameterized,
TypedStack having to be both abstract and parameterized, we introduce the metaclass AbstractTypedClass, which is a subclass of both AbstractClass and TypedClass, and therefore an instance of MIClass. Conflicting methods, namely new (and new!), should be redirected to AbstractClass.

MIClass newName: #AbstractTypedClass
  superclass: 'AbstractClass TypedClass'
  instanceVariableNames: "
  category: 'MetaClass-Combination'"

AbstractTypedClass methodsFor: 'conflicting methods' new
  ^self AbstractClass.new!.

AbstractTypedClass newName: #TypedStack
  superclass: Stack
  instanceVariableNames: "
  category: 'Stack-Collection'"

TypedStack methodsFor: 'operations'
push: x
  (x isKindOf: self class type)
    ifTrue: [super push: x]
    ifFalse: [self error: 'wrong type']

TypedStack newName: #IntegerStack
  superclass: TypedStack
  instanceVariableNames: "
  type: Integer
  category: 'Stack-Collection'"

9 Class/Metaclass Module vs Uniform Creation

9.1 Limitations of the Classtalk Library

The Smalltalk-80 class/metaclass module is split by Classtalk into two explicit components. On the one hand, this allows an unlimited level of metaclasses and provides the user with more freedom. But on the other hand, we need to define (meta)metaclasses to define extended class creation methods each time we add some new instance variable, e.g. metaclasses MetaTypedClass and MetaAccessClass.

The class/metaclass module remains necessary when defining extended creation messages, as in standard Smalltalk-80. But Smalltalk-80 takes care of implicitly creating a metaclass to support the class method, whereas in Classtalk the programmer has the burden to explicitly defining the class method.

Another limitation of non-uniform creation is illustrated by next example. We want to model classes whose all instance variables are public. Therefore we define

PublicClass as an instance of AutoInitClass and a subclass of AccessClass. Its init method generates accessors for all instance variables:

AutoInitClass
  newName: #PublicClass
  superclass: AccessClass
  instanceVariableNames: "
  category: 'MetaClass-Library'"

|PublicClass methodsFor: 'init'
|  self makeAccessOn: instanceVariables!

Unfortunately this scheme does not work. The method init is called during the process of allocation (method new redefined in AutoInitClass) and before creation of the class (method newName:...category:). Consequently instanceVariables is as yet initialized (value nil) and no accessing method is generated.

A solution is to redefine newName:...category: in order to call the init method. But init will be called twice (once at allocation time and once at creation time), because of non uniformity.

In summary, programming with explicit metaclasses requires an uniform creation protocol.

9.2 Uniform Creation in Classtalk

Uniform creation, method create:, is defined as the combination of standard allocation (basicNew) and a generic uniform initialization initialize:. In order to be usable by all classes, Smalltalk-80 or Classtalk ones, create: is defined by Behavior:

|Behavior methodsFor: 'creation'
|  create: initArray
    ^self basicNew initialize: initArray!

There are two initialization methods: one for (meta)classes, owned by Classtalk, and another for objects, defined by Object. Initialization of classes specializes initialization of general objects (use of pseudo-variable super):

|Classtalk methodsFor: 'initialization'
|  initialize: initArray
    super initialize initArray.
    self environment: Smalltalk
      variable: false
      words: true
      pointers: true
      category: category!

The method environment:...category: is defined as equivalent to the method...
9.3 General Initialization
The method initializeInstance: owned by Object initializes instance variables of every object. Because their names and number is defined for each class, this method should accept a variable number of arguments. Unfortunately Smalltalk-80 syntax does not allow selectors with variable arity. Therefore, we need to group the arguments into a single data structure, such as array. The creation of a cartesian complex would look like:

Cartesian create: #(y 2 x 1)

This follows the strategy of CommonLisp-like keywords, which may be reordered at will, as opposed to explicit and ordered keywords in Smalltalk-80.

9.4 Implementation
The main problem is to evaluate the arguments associated to instance variables.

One solution is to extend Smalltalk-80 syntax in order to support dynamic creation of arrays, by using some macro-method or macro-character analog to Lisp's backquote.

Another solution is to evaluate the arguments through explicit calls to the compiler. For each instance variable, the standard method instVarAt:put: assigns the variable with the value computed by the compiler:

!Object methodFor: 'initialize-release'
Initialize: initArray
  | i max ivNames aContext aCompiler |
  initArray isNil ifFalse:
    [i _ 1,
      max _ initArray size,
      ivNames _ self class allInstVarNames,
      aContext _ thisContext sender sender,
      aCompiler _ Compiler new,
      [i < max] whileTrue:
        [self instVarAt: (ivNames indexOf: (initArray at: i))
          ifAbsent: [self error:
            'unknown instance variable: ' ', (initArray at: i) printString]]
          put: (aCompiler
            evaluate: (initArray at: i+1) printString
            in: aContext
to: aContext receiver
            notifying: self
            ifFall: [self error:
              'compilation of initialize failed'])].
    i _ i+2]
]

9.5 Class talk Library Revisited
We redefine the metaclass TypedClass and its instance IntegerStack to show this simplification. Defining MetaTypedClass is no more necessary:

Class talk create: #(
  name #TypedClass
  superclass Class talk
  instanceVariables 'type'
  category 'MetaClass-Library'
)

TypedClass create: #(
  name #IntegerStack
  superclass TypedStack
  instanceVariables ""
  type Integer
  category 'Stack-Collection'
)

The good version of PublicKey uses a redefinition of the initialize: method. AutoInitClass is no more necessary:

Class talk create: #(
  name #PublicKey
  superclass AccessClass
  instanceVariables ""
  category 'MetaClass-Library'
)

PublicKey methodFor: 'init'
Initialize: initArray
  super initialize: initArray,
  self makeIVAccessOn: instanceVariables !

10 Future Work
Experimenting with Class talk revealed the following limitations:

Methodology
The Smalltalk methodology suggests to define examples of a class as class methods. Class talk metaclasses are no longer implicitly private to a class. Consequently we need to provide another approach, for example by adding an instance variable at the class level.

Class/Metaclass Compatibility
Defining explicit metaclasses raises the issue of compatibility between a class and its metaclass, i.e. the mutual hypotheses about the instance variables and methods they define [Graube89]. This may lead to non-trivial problems when reusing standard Smalltalk-80 classes. For instance, if defining Stack as a subclass of OrderedCollection OrderedCollection defines the private initialization method setIndices. The allocation method of OrderedCollection class is redefined in order to automatically ensure the initialization:

430 COPSLA '89 Proceedings October 1-6, 1989
[Bornino&OSheaf87] A. Bornino and T. O'Shea, Deltatalk: An Empirically and
Aesthetically Motivated Simplification of
the Smalltalk-80 Language, ECOOP'87,
LNCS, No 276, pages 1-10, Springer-
Verlag, June 1987.
[Briot&Cointe87] J.-P. Briot and P. Cointe, A
Uniform Model for Object-Oriented
Languages Using The Class Abstraction,
[Cointe87] P. Cointe, Metaclasses are First
Class: the ObjVlisp Model, OOPSLA'87,
pages 156-167.
[Cointe&Graube88] P. Cointe and N. Graube,
Programming with Metaclasses in CLOS,
First CLOS Users and Implementors
Workshop, Xerox Parc, Palo Alto CA, USA,
[Cointe88] P. Cointe, A Tutorial Introduction
To Metaclass Architectures as Provided
By Class Oriented Languages,
International Conference on Fifth
Generation Computer Systems (FGCS'88),
Vol. 2, pages 592-608, Ipot, Tokyo, Japan,
[Goldberg&Robson83] A. Goldberg and D.
Robson, Smalltalk-80: the Language and
its Implementation, Series in Computer
Science, Addison Wesley, 1983.
[Graube89] N. Graube, Metaclass
Compatibility, in same volume.
[Ingalls&Borning82] D.H.H. Ingalls and A.H.
Bornino, Multiple Inheritance in
Smalltalk-80, Proceedings of the National
Conference on Artificial Intelligence,
pages 234-237, USA, August 1982.
[Malenfant&al89] Malenfant, G. Lapalme and
J. Vaucher, ObjVPprolog: Metaclasses in
Logic, ECOOP'89, Cambridge University
[Ungarro&Smith87] D. Ungaro and R.B. Smith,
Self: The Power of Simplicity, OOPSLA'87,
pages 227-242.
[Wolinski89] F. Wolinski, Le Système MY2C:
Modélisation et Génération d' Interfaces
Homme-Machine, Report 89/38, Latora,
Université Pierre et Marie Curie, Paris,
April 1989.

Conclusion
In this paper we pointed out the
limitations of the metaclass architecture of
Smalltalk-80. We introduced explicit
metaclasses and uniform creation a la
ObjVlisp to alleviate these problems. The
resulting system provides a platform to
experiment and apply metaclass-oriented
methodology with the help of the Smalltalk-
80 libraries and environment.

We thank Francis Wolinski for providing
his generic tree editor that we interfaced
with the ClassTalk environment.

Bibliography
Boscotrecase, T. Flagella and M. Gaspari,
Metalevel Programming in CLOS,
ECOOP'89, Cambridge University Press,
July 1989.
[Bobrow&Kiczales88] D.G. Bobrow and G.
Kiczales, The Common Lisp Object System
Metaobject Kernel: A Status Report, ACM
Conference on Lisp and Functional
October 1-6, 1989
OOPSLA '89 Proceedings
431