Modelling Objects in PICT

Jean-Guy Schneider, Markus Lumpe
Software Composition Group

IAM-96-004
January 1996

Abstract

For the development of present-day applications, programming languages supporting high order abstractions are needed. These high order abstractions are called components. Since most of the currently available programming languages and systems fail to provide sufficient support for specifying and implementing components, we are developing a new language suitable for software composition. It is not clear how such a language will look like, what kind of abstractions it must support, and what kind of formal model it will be based on. Object-oriented programming languages address some of the needs of present-day applications, and it is therefore obvious to integrate some of their concepts and abstractions in the language. As a first step towards such an integration, we have to define an object model. Since no generally accepted formal object model exists, we have chosen the π-calculus as a basis for modelling. In order to find a suitable object model, we have built up an object modelling workbench for PICT, an implementation of an asynchronous π-calculus. In this work, we define a first abstract object model, describe several implementations of the object model in PICT, and discuss interesting features and possible extensions.

Keywords: π-calculus, PICT, object modelling, software composition.

CR Categories and Subject Descriptors: D.1.3 [Programming Techniques]: Concurrent Programming; D.1.5 [Programming Techniques]: Object-oriented Programming; D.2.1 [Software Engineering]: Requirements/Specification; D.3.3 [Programming Languages]: Language Constructs and Features; F.4.3 [Mathematical Logic and Formal Languages]: Formal Languages.

Authors' address: Institute for Computer Science and Applied Mathematics (IAM), University of Berne, Neubrückstrasse 10, CH-3012 Bern, Switzerland; e-mail: {schneidr,lumpe}@iam.unibe.ch; WWW: http://iamwww.unibe.ch/~scg.

1 Introduction

Software development and maintenance has always been an expensive task. To reduce its costs, special-purpose programming languages and methods have been developed and used. For present-day applications however, which are getting more complex and increasingly open, these methods and languages are not suitable any more; new ones have to be developed that address open systems requirements.

Object-oriented programming addresses some of the needs of present-day applications, but only offers limited support for viewing applications as configurations of adaptable and reusable software components.

What we need are higher abstractions than classes and objects. These higher abstractions are components, being configurable entities which can be composed to build an application [ND95]. Unfortunately, most object-oriented techniques fail to provide suitable abstractions for general component specification and component composition (composition mechanisms) [NM95]. In order to get a system for composition, where components can be specified and implemented, but also components written in other systems/languages can be used, we have to define our own composition language. In this language, we would like to integrate and combine aspects, concepts and paradigms of existing languages and systems and develop an abstract model for software composition [NGT92, NM95]. Since a lot of new applications run in distributed environments, our model for software composition must also support the definition and use of concurrent components.

According to [Nie92], the development of concurrent object-based programming languages has suffered from the lack of any generally accepted formal foundation for defining their semantics. Several formal models have been presented (refer to [Men94] for a summary), but none of them has been used as a formal basis for modelling software composition.

Unfortunately it is not clear, what kind of formal model is suitable for software composition. Therefore we have to define and implement several models, which need to be evaluated.

The π-calculus is a calculus in which the topology of communication can evolve dynamically during evaluation [Mil89]. It has been successfully used to model objects [Jon93, BS95a] and simple object-oriented programming languages [Wal95]. Therefore it seems to be a good formal foundation for modelling software composition, especially since it possible to embed the λ-calculus into the π-calculus, where methods for program reasoning have been developed. Fortunately there exists an implementation of an asynchronous π-calculus [Pie95b], which has already been used for the implementation of a simple object model [PT95]. This object model lacks several abstractions available in most object-oriented languages, but can be used as a basis for further modelling. Based on this model, we try to find other implementations of objects in PICT, in order to obtain an object modelling workbench. Using this workbench, it will be easier to evaluate and implement object models which can be used as a basis for a composition language. As a first step towards a modelling workbench, we will only concentrate on modelling common abstractions used in object technology, which are not necessarily concurrent. Full concurrency will be introduced in a later stage, but we will already discuss problems and possible solutions due to a concurrent object model.

In this work, we first define an abstract object model based on object models used in other programming languages (section 2). In section 3, we describe several implementations of our model and cite interesting features of PICT. Finally in section 4 we summarize our work and mention future possible work.

2 Object Model

As mentioned above, it is not clear what kind of object models are suitable for software composition. In order to be able to evaluate several object models, we think that the modelling workbench should contain features of already existing object models. As a starting point, we have defined a first abstract object model, which will be used throughout the rest of this work. It has been strongly influenced by the object models of C++ [Str94], Eiffel [Mey92], and Object Pascal [BF95], and supports the following features:

- class variables,
- class methods,
- instance variables,
- instance methods,
- self-reference of objects,
- inheritance,
- genericity,
- static and dynamic binding.

To keep the first model simple, methods and variables of a class are either exported to clients (public) or hidden (private); no selective export like in Eiffel will be modelled. A subclass of a given class will inherit all public features[1] and private instance variables from its superclass, but none of the private methods. A subclass can redefine any of its inherited public features. Although it might be necessary to model multiple inheritance, we will not discuss problems due to this mechanism.

Since we do not know yet what kind of features are necessary for an object model for software composition, it is possible that we have to add other features to our list. They will be added and modelled during evaluation.

In Figure 1, the interfaces of two example classes are described in pseudo-code notation which will be used throughout the rest of this work to explain our object modellings in PICT. The example classes use most of the features introduced above. The class IntStack defines a class for integer stacks with its usual features. The keyword public is used to define features of a class which are exported to clients, whereas private denotes features only visible within the class. common is used to define features shared by all objects of a class (class variables and methods), and override is used to specify an inherited feature which will be redefined. The class IntStack has a class variable Pushed, which is used to count the number of items pushed on all stacks. The class IntTower is a subclass of IntStack, where it is only possible to push items in a decreasing order. A description of both classes can be found in Appendix A or in [BL94].

[1] The term feature is used like in Eiffel; it denotes methods and variables of a class.

```
class IntStack = {
        private
                LocalPushed : Integer;
                Contents : List of Integer;
        public
                function empty () : Boolean;
                procedure push ( Value: Integer );
                procedure pop ();
                function top () : Integer;
                function localPushed () : Integer;
        private
                Pushed : Integer; common;
        public
                function pushed () : Integer ; common;
}

class IntTower = IntStack {
        private
                function CanPush ( Value: Integer ) : Boolean;
        public
                procedure push ( Value: Integer ); override;
}
```

Figure 1: Abstract object model.

3 Implementation in PICT

In this section, we will describe several PICT implementations of the abstract object model. Readers not familiar with the language PICT can find an introduction in [Pie95b] or [Var96]. For additional background in the π-calculus, please refer to [Mil89, Mil91]. Details about the implementation of PICT, especially its type system, are described in [Tur96].

3.1 Object model of Pierce and Turner

In [PT95, Pie95b, Tur96], Pierce and Turner introduce a simple object model based on processes: an object is a process consisting of

- a server process with some internal state, being able to service requests of clients to query and manipulate the state, and
- a set of request channels, which can be used by clients to request services.

The following process based reference cell [Tur96] illustrates their object model:

```
def ref [init] =
    let
        new Contents
        run Contents!init
    in
        set = abs [v,c] > Contents?_ > (Contents!v | c![]) end,
        get = abs [r] > Contents?v > (Contents!v | r!v) end
    end
```

A process based reference cell consists of an internal channel `Contents`, which is used to store the internal state of the object, and two request channels (`set` to set a new state and `get` to read the current state). The initial state of the reference cell is set by the `init` parameter. In order to protect against other processes reading and writing `Contents`, its declaration and initialization is wrapped in a local block.

Each request channel is the interface to another process. Theses processes are defined as anonymous process abstractions (using the keyword `abs`), and are the only processes being able to query and manipulate the state of the object. In order to simplify their use, they are packed in a record.

The `let ... in ... end` construct defines a process *template*, which can be used by the function `ref` to create reference cell objects[2]

```
val cell = ref [50]
```

Requests to an object are performed by the usual dot notation:

```
run cell.set[20]; prInt [cell.get[]];
```

The reader may have noticed that are no explicit type annotations in the reference cell example. The correct types are inferred by the type inference algorithm included in PICT.

As a first approach, we will base our modelling on the object model described in this section and extend it wherever needed.

3.2 Model 1

In Figure 2, the complete PICT code of our first implementation is described. Like in the simple object model of Pierce and Turner [PT95, Pie95b], an object is modelled as a process, instance variables are modelled as local channels, and methods are defined as a record of process abstractions. For list processing and exception handling, predefined libraries are used [Pie95a]. Although the implementation only works for integers, it can be easily extended for any kind of objects by introducing a generic parameter [Pie95b]. Figure 9 for an example)

In order to model the class variable `Pushed`, a global channel `Pushed` is defined and initialized to `0`. Each stack object gets this channel as a parameter at creation and can use it as

[2] The process `ref` can be seen as a reference cell factory.

```
def IntStack [Pushed: ^Int] =
    let
        new LocalPushed
        run LocalPushed!0
        new Contents: ^(List Int)
        run Contents ! (nil[])

        def empty [res] > Contents?aList > (Contents!aList | res!(null[aList]))
    in
        record
            empty = empty,
            push = abs [v,c] > Contents?aList > LocalPushed?v > Pushed?g >
                        (Contents!(cons [v, aList]) |
                         LocalPushed!(v+1) | Pushed!(g+1) | c![]) end,
            pop = abs [c] >
                        (if (empty[]) then
                            raise![exitOnExn, "Pop on empty stack!",c]
                        else
                            Contents?aList > LocalPushed?v > Pushed?g >
                            (Contents!(cdr[aList]) | LocalPushed!(v-1) |
                             Pushed!(g-1) | c![])
                        end) end,
            top = abs [t] >
                        (if (empty[]) then
                            raise[exitOnExn, "Top on empty stack!"];
                        else
                            Contents?aList >
                            (Contents!aList | t!(car[aList]))
                        end) end,
            localPushed = abs [l] > LocalPushed?v > (LocalPushed!v | l!v) end
        end
    end

new Pushed                                          {- class variable -}
run Pushed!0
def pushed [g] > Pushed?v > (Pushed!v | g!v)        {- class method -}
```

Figure 2: Source code of first model.

if it were an instance variable. Of course, any other process can also modify the contents of this channel, and therefore we do not have the kind of data encapsulation needed.

The call of `pop` or `top` is only valid if the stack is not empty; on a call on an empty stack an exception is raised, and the program terminates. Both methods have to check whether there is at least one item pushed onto the stack before they can proceed. Since stack objects already have a function `empty`, it is obvious to use this function within `pop` and `top`. Because the simple object model of Pierce and Turner does not local method calls, the implementation of `empty` has to move to the declaration part of `IntStack`: a function `empty` is defined within the scope of `IntStack`. Due to the use of different name spaces in PICT, it is possible to have a label and a process with the same name in the scope of a record. The request channel `empty` forwards its requests directly to the (local) process of the same name.

```
def IntStack [Pushed: ^Int] =
    let
        ...
    in
        record
            ...
        end
    end

def IntStackClass [] =
    let
        new Pushed
        run Pushed!0
    in
        record
            pushed = abs [g] > Pushed?v > (Pushed!v | g!v) end,
            Create = abs [] = IntStack [Pushed] end,
            GetClassName = abs [] = "IntStack" end
        end
    end
```

Figure 3: Source code of second model.

Stack objects can be created by

```
val s1 = IntStack[Pushed]
```

methods are called by the usual dot notation

```
run s1.push[5]; prInt[s1.localPushed[]];
```

Several abstractions are not supported by this first model. For example it is not possible to model self-reference of objects (which is needed for the support of dynamic binding and the local call of exported methods), and there is no possibility to express inheritance. This approach does not encapsulate common class features: class variables and methods are defined and implemented in the global scope. The following models will discuss possible solutions of these problems.

3.3 Model 2

Throughout the rest of this section, we will not show the complete PICT code of our implementations, but only the major differences to the previous versions.

In order to make the channel `Pushed` only visible to integer stacks, it is not correct to define it in the declaration part of `IntStack`. Each stack would get its own nonshared instance variable `Pushed`, which is not the intended behaviour if we want to model class variables. To solve the problem of class variables and methods, we have to find another solution.

On of the main ideas of CLOS [Pae93, KdRB91], Smalltalk [GR89], and other recent publications [BS95b, Chi95] is the definition of classes as *metaobjects* and the introduction

of *metaobject protocols*. For the purpose of modelling objects in PICT, we currently do not intend to introduce a full metaobject protocol. The metaobject defined by `IntStackClass` described in Figure 3 is only responsible for the declaration and initialization of class variables, definition of class methods, and creation of instances of the class. The process abstraction `IntStack` is defined the same way as in the previous model.

Before objects of the class IntStack can be instantiated, the corresponding metaobject has to be created:

```
val IntStackMetaObj = IntStackClass[]
```

Stack objects are instantiated by

```
val s1 = IntStackMetaObj.Create[]
```

By introducing metaobjects, we are now able to model class methods and class variables for the corresponding class. Since PICT is statically typed, not only instances of a class, but also the corresponding metaobject and all its methods are well typed (not like in CLOS or Smalltalk, where metaobjects are untyped). Therefore, the `IntStackMetaObj` metaobject generates integer stacks (as desired). If we would like to have generic stacks, we have to introduce a generic parameter for `IntStackClass` (refer to section 3.9). The only other change is the redefinition of `Create` in `IntStackMetaObj`:

```
Create = abs [:T:][] = Stack [:T:][Pushed] end
```

Now it is possible to create stacks with items of a different type, like

```
val s1 = StackMetaObj.Create [:String:][]
```

for string stacks. The class variable `Pushed` still counts the items pushed on all stacks, whatever their type of items is. In section 3.9 we will show the implementation of a generic stack class.

3.4 Model 3

If we want to introduce an pseudo instance variable `Self`[3], which is the self-reference to an object, `Self` must have a type which corresponds to the type of the object. Since PICT uses different name spaces for types and process abstractions, the type for the class `IntStack` is also called `IntStack`. Like all instance variables, we will model `Self` as a local channel. Therefore, `Self` is of type `^IntStack` (channel of `IntStack`).

As a first attempt, one could try to model self-reference as a parameter at object creation (like `Pushed`):

```
val s1 = IntStackMetaObj.Create [s1, ...]
```

[3] This pseudo-variable is comparable to this in C++, Current in Eiffel, and self in Smalltalk.

```
def IntStack [Pushed: ^Int, MetaInfo: IntStackClass] : IntStack =
    let
        val MetaObj = unfold (MetaInfo)
        new Self: ^IntStack
        ...
    in
        (fold IntStack record
            ...
            setSelf = abs [s,c] > (Self!s | c![]) end
        end)
    end

def IntStackClass [] : IntStackClass =
    let
        new Self: ^IntStackClass
        ...
    in
        (fold IntStackClass record
            Create = abs [NewObj] > Self?Meta > Self!Meta |
                        (let
                            val obj = IntStack [Pushed, Meta]
                         in
                            (unfold obj).setSelf [obj]; NewObj!obj
                        end) end,
            setSelf = abs [s,c] > (Self!s | c![]) end,
            ...
        end)
    end
```

Figure 4: Source code of third model.

This does not work in PICT because it is not possible to use the value defined in a value abstraction recursively. Therefore the self-reference has to be assigned *after* the object has been created:

```
val s1 = IntStackMetaObj.Create [...]
run s1.setSelf [s1];
```

The method `setSelf` assigns the value of the object to its pseudo-variable `Self` (Figure 4).

In all previous models, we did not have to explicitly specify the type of `IntStack` nor `IntStackClass`; it is inferred by the type inference system of PICT. If we want to use `Self` within instance methods, its exact type has to be known at the position where it is used. If we do not give an explicit type annotation for `Self`, the type inference algorithm cannot infer the correct type; an explicit type annotation is required. Since `IntStack` has

a method `setSelf` with a parameter of type `IntStack`, the type of `IntStack` is recursive

```
type IntStack =
    Rec(T) Record
        push: ![Int,Sig],
        pop: ![Sig],
        top: ![!Int],
        empty: ![!Bool],
        localPushed: ![!Int],
        setSelf: ![T,Sig]
end
```

and the pseudo-variable `Self` is of type `^IntStack` (Figure 4). Since we also need to model self-reference of the metaobject, the metaobject type is defined accordingly:

```
type IntStackClass =
    Rec(T) Record
        pushed: ![!Int],
        Create: ![!IntStack],
        GetClassName: ![!String],
        setSelf: ![T,Sig]
end
```

Due to the fact that both `IntStack` and `IntStackClass` are recursive types, the programmer has to take care of correct *folding* and *unfolding* of entities of both types [Pie95a]: *unfold* transforms a value with a recursive type in one with a nonrecursive type, *fold* is used for the inverse transformation. The major drawback of recursive types is that no subtype relation can be established between them: two recursive types `A` and `B` are either equal or noncomparable[4]. Since this is not suitable for modelling inheritance (where we need subtype relations), recursive types must be omitted. This will be a topic of the next section.

Another disadvantage of this model is that the method `setSelf` is exported, which in fact should not happen. It is possible to abuse `setSelf` like in

```
val s1 = IntStackMetaObj.Create[]
val s2 = IntStackMetaObj.Create[]
run s1.setSelf[s2];
```

which is probably not the way `setSelf` should be used.

3.5 Model 4

The previous model (Figure 4) has two major disadvantages: export of `setSelf` and use of recursive types. With the model introduced in this section we will be able to get rid of most of these disadvantages.

The method `setSelf` has two preconditions: first it must be applied as the first method to the object, and second it has to be used with the object itself as the one and only

[4] According to Pierce, this will likely be changed in a future version of PICT.

```
def IntStack [Pushed: ^Int, MetaObj: IntStackClass] : IntStackImpl =
    let
        new Self: ^IntStack
        ...
    in
        record
            ...
            setSelf = abs [s,c] > (Self!s | c![]) end
        end
    end

def IntStackClass [] : IntStackClassImpl =
    let
        new Self: ^IntStackClass
        ...
    in
        record
            Create = abs [NewObj: !IntStack] > Self?Meta > Self!Meta |
                        (let
                            val obj = IntStack [Pushed, Meta]
                          in
                            obj.setSelf [obj]; NewObj!obj
                        end) end,
            ...
        end
    end
```

Figure 5: Source code of fourth model.

parameter. Other usages are forbidden. The model described in section 3.4 does not guarantee the correct behaviour when `setSelf` is abused. In order to prevent abuse, one could think of changing the implementation of `IntStack` by using *choice* and *event channels* [Pie95b] in a way that `setSelf` is only accepted as the first method call. This would guarantee the first precondition, the second still remains open. What we need is a construct where either the use of more than one `setSelf` causes a *compile time* error or `setSelf` is not used any more. In the next section we will show how this is possible.

As a first approach for preventing the abuse of methods like `setSelf` and increasing the possibilities of data encapsulation, we introduce the notion of *interface* and *implementation* type. In the interface type, all features meant to be used by clients of an object are specified. The implementation type extends the interface by implementation details only used by the object itself and its corresponding metaobject. Private methods of a class are not part of the implementation type.

The interface and implementation type of `IntStack` and `IntStackClass` are defined as follows:

```
type IntStack =                              {- Interface Type -}
    Record
        push: ![Int,Sig],
        ...
        localPushed: ![!Int]
end

type IntStackImpl = IntStack                 {- Implementation Type -}
    with
        setSelf: ![IntStack,Sig]
end

type IntStackClass = MetaObject IntStack     {- Interface Type -}
    with
        pushed: ![!Int]
end

type IntStackClassImpl = IntStackClass       {- Implementation Type -}
    with
        setSelf: ![IntStackClass,Sig]
end
```

The `with` operator extends a given `Record` type with new fields. If such a field already exists, it is overwritten by the new one. As a step towards a modular programming environment, a generic metaobject type has been introduced, where the common features of all metaobjects are combined:

```
type MetaObject T =
    Record
        Create: ![!T],
        GetClassName: ![!String]
end
```

The `Create` method of the metaobject (Figure 5) creates an object of type `IntStackImpl`, and can therefore use the method `setSelf`. The result channel `NewObj` on the other hand is of type `IntStack`, which is a supertype of `IntStackImpl`. Since it is possible to send along a channel a value of a subtype of the actual channel type, it is safe to send an object of type `IntStackImpl` along a channel of type `IntStack`. Although `Create` creates an object of type `IntStackImpl`, the static type of the created object is `IntStack`, and only methods defined for this type can be used. The implementation part of the object has simply been cut off. This is the first time we use *type restriction* for improving data encapsulation.

The main reason why a recursive type had to be introduced in the previous model was the parameter of the method `setSelf`, which had to be equal to the type of the object itself. By changing the type of the parameter to `IntStack`, the type of `IntStackImpl` is not recursive any more, and the value represented by the channel `Self` is of type `IntStack`. With this solution, the usage of `Self` is restricted only to features of the interface type.

Another major problem has still not been solved: we have no chance to check whether the value passed to an object using `setSelf` is identical to the object itself. In order to guarantee the second precondition (refer to Page 11), we must look for another solution.

```
def IntStackClass [] : IntStackClass =
    let
        val MetaSelf = emptyRef [:IntStackClass:][]
        ...
        def Create [] : IntStackImpl =
            let
                val Self = emptyRef [:IntStackImpl:][]
                val NewInstance =
                    let
                        val LocalPushed = ref[0]
                        val Contents = ref [nil[]]
                    in
                        record
                            pop = abs [c] > if ((Self.get[]).empty)
                                        then ...
                            ...
                            MetaObj = MetaSelf  {- Reference Cell of Meta Object -}
                        end
                    end
            in
                Self.set[NewInstance];
                Self.get[]
            end
        ...
    in
        ...
    end
```

Figure 6: Source code of fifth model.

3.6 Model 5

The models described in the previous sections still have several problems, which we will solve in this section. We will introduce a much more convenient way for modelling self-references, which also leads to a much better data encapsulation.

As an improvement, the process definition `IntStack` is not in the file scope any more, but has moved to the declaration part of the metaobject. Now it is not possible any more to create stacks without using the `Create` method of the metaobject.

The first simple object which is modelled in [Pie95b] is a *reference cell* (refer also to section 3.1). A reference cell is an updatable data structure [Tur96]. By using the `set` method, a new value can be assigned to the reference cell; the `get` function returns the value actually stored. Although a reference cell uses a local channel to store the value, the exact implementation is not of primary concern to us. Reference cells can either be created by using

`val` *cell* `= ref [:`*type*`:][`*initVal*`]`

where a initial value is set, or

`val` *cell* `= emptyRef [:`*type*`:][]`

where no initial value is assigned.

As shown in Figure 6, the `Create` method of the metaobject has changed. In the declaration part, an *empty* reference cell (`Self`) and an object `NewInstance` of type `IntStackImpl` are created. The value of `NewInstance` is then assigned as the (first) contents of the reference cell. As the result of `Create`, the *contents* of the reference cell (the newly created object) is returned. Since the name `Self` and its type is known, it can be used within the declaration of `NewInstance`. `Self.get[]` is nothing other than an alias to `NewInstance`, and can be used as the self-reference instead. The self-reference cannot be changed any more, because it is not possible to access the reference cell `Self` from outside the object. The same encoding is also used to obtain a self-reference of the metaobject (not shown in Figure 6). This solution has two major advantages: an elegant modelling of self-references and an easy way to make the distinction between the internal representation of an object and its exported interface.

Although the `Create` method defined in the declaration part of `IntStackClass` creates an object of type `IntStackImpl`, the exported `Create` method returns an object of type `IntStack`. Again type restriction is used to hide the implementation part of an object.

The modelling of instance variables has also changed: reference cells are used instead of local channels. This has the advantage of making them easier to use, but it means that we can no longer synchronize concurrent accesses. Various approaches are possible for avoiding interferences and deadlock, but a detailed investigation will be a topic of future work. For some preliminary results in modelling concurrency control, refer to [Var96].

3.7 Model 6

One of the main features of object oriented programming is inheritance. Usually one distinguishes between *interface inheritance* and *code inheritance*. Interface inheritance is not a problem in PICT since it is possible to extend a given record type with new fields. We have already used this construct in previous models (`IntStackImpl` inherits the interface of `IntStack`). Code inheritance is more difficult. What we need is an abstraction where a *subclass* of a given class can reuse the code implemented in its superclass. In the previous model, all process abstractions are defined within the scope of `IntStack`. No other process has had the possibility to access the implementation. The data encapsulation we use is too restricted; it needs to be opened.

In order to understand one possibility to export the implementation of a process abstraction, we have to investigate the translation of the process definition abstraction `def` into core PICT. A process `name` defined as

```
def name [par] > proc
```

is translated into

```
new name
run name?*[par] > proc
```

Since process abstractions are translated into channels and channel names are values, it is possible to pass a process abstraction to another process. By introducing a method `getMethods`, each object can send its private process definitions to other objects. This is not a very convenient solution since the exact type of `getMethods` has to be specified,

```
type IntStackClass = MetaObject IntStack  {- Interface Type -}
    with
        pushed: ![!Int],
        Pushed: Ref Int
end

type IntStackImpl = IntStack              {- Implementation Type -}
    with
        MetaObj: Ref IntStackClass,
        Contents: Ref (List Int),
        LocalPushed: Ref Int
end

...

def IntStack'pop [Self: IntStackImpl] =   {- Method Definition in File Scope -}
        if (Self.empty[]) then
            raise [exitOnExn, "Pop on empty Stack!"]
        else
            let
                val Stk = Self.Contents.get[]
                val Pushed = (Self.MetaObj.get[]).Pushed
            in
                Self.Contents.set [cdr [Stk]];
                Self.LocalPushed.set [Self.LocalPushed.get[] - 1];
                Pushed.set[Pushed.get[] - 1]
            end
        end
```

Figure 7: Source code of sixth model.

which usually is not trivial and results in redundant type annotations.

Due to the fact that process definitions have to be opened anyway, an easier solution is chosen. Each class declares its instance variables as reference cells in its implementation type (Figure 7). All exported methods are defined globally, with a parameter of the corresponding implementation type as an additional first parameter. This is the typical implementation of C++ methods. We use *name mangling* to unambiguously specify the method name[5]. The external object interface is not affected by this modification. Within a method, it is necessary to access instance variables or call other instance methods by using the self-reference. This is the reason why instance variables have to be defined in the implementation type. Objects can use any of the globally defined methods if their implementation type is a *subtype* of the first parameter of the feature. This is important for modelling inheritance.

An object's metaobject information is stored in a reference cell. Therefore it is possible to change the metaobject of an object, which is necessary when objects should change their type at runtime. This is often used in prototype based languages [Lie86]. Note that it is only possible to change an object's metaobject when the type of the new metaobject is a subtype of the object's metaobject type.

[5] We have chosen `ClassName'methodName` to name methods.

This model has an additional property: it is the first model supporting *dynamic binding*. Dynamic binding is only available for exported methods, but this is not a restriction, because there is no need to bind private methods dynamically. With this model, we can use both static and dynamic binding. In order to use dynamic binding, a method has to be called through an object (Smalltalk terminology: sending a message to an object). To force a method to be bound statically, we have to call this method directly without an object. The following code fragment shows the static and the dynamic binding of the method `push` of the class `IntStack`:

```
val s1 = IntStackMetaObj.Create[]
...
IntStack'push[s1,1];[6]        {- static binding -}
...
s1.push[1];                    {- dynamic binding -}
```

This example needs some explanations. In order to obtain static or dynamic binding, different call mechanisms have to be used. For static binding the method is called directly, and for dynamic binding the method is selected through the exported interface of the object.

In order to understand how dynamic binding works, we take a closer look at the implementation of C++, where a method table is used to select virtual methods. This table contains pointers to all virtual methods which are defined (inherited, redefined or new defined) for the class. Each object has an unnamed hidden instance variable which belongs to the virtual method table of the class [Bor91]. The user has no access to this instance variable. The value of this instance variable is set in the constructor(s) of the class.

All object models we have defined so far do not use an explicit method table. In fact, these models do not support dynamic binding. But we have a comparable construct which can serve as method table – the interface record.

In the interface record, the public interface of an object is encoded. Only public methods can be bound dynamically. Like in C++, private methods are always bound statically. The interface record initialization hides the name mangling of the methods and provides the user with the original call interface for the methods:

```
record
    empty = abs [] = IntStack'empty[Self.get[]] end,
    push = abs [v] = IntStack'push[Self.get[],v] end,
    ...
end
```

The anonymous process abstractions used to initialize the record values play roughly speaking the role of pointers to the methods. Therefore we have a comparable implementation of a virtual method table in PICT. Moreover, this implementation has an advantage over the C++ solution. The current object `Self` is bound to the first parameter within the initialization of the record fields. There is no need to provide the current object as first

[6] This only works if `s1` is of type `IntStackImpl`.

parameter every time a method is called (like in the C++ implementation). Therefore, the call interface of the methods remain the same according to their definition:

```
procedure push ( Value: Integer );
```

is translated into

```
push = abs[v] = IntStack'push[Self.get[],v] end
```

which can be used like

```
aStackObj.push[2];
```

3.8 Model 7

In all the previous models we tried to obtain a sufficient data encapsulation. With the model presented in this section, we finally start to model inheritance.

In Figure 1 we have introduced a class IntTower, which is *subclass* of IntStack. As we will see in this section, the corresponding type `IntTower` is also a *subtype* of `IntStack`.

Before the implementation of IntTower can be modelled, the corresponding types have to be defined:

```
type IntTower = IntStack                    {- IntTower <: IntStack -}

type IntStackClass = MetaObject IntStack
    with
        pushed: ![!Int],
        Pushed: Ref Int
end

type IntTowerClass = MetaObject IntTower
    with                          {- IntTowerClass <: IntStackClass -}
        pushed: ![!Int],
        Pushed: Ref Int
end

type IntStackImpl = IntStack
    with
        LocalPushed: Ref Int,
        Contents: Ref (List Int),
        MetaObj: IntStackClass
end

type IntTowerImpl = IntStackImpl   {- IntTowerImpl <: IntStackImpl -}
    with
        MetaObj: IntTowerClass
end
```

Since the class `IntTower` has exactly the same interface as the class `IntStack`, the interface type `IntTower` is equal to the type `IntStack` (and also a subtype of `IntStack`). Both classes have the same class variables and class methods, and therefore the corresponding

```
def IntTowerClass [] : IntTowerClass =
    let
        ...
        def Create [] : IntTowerImpl =
            let
                val Self = emptyRef [:IntTowerImpl:][]
                val NewInstance =
                    let
                        val LocalPushed = ref[0]
                        val Contents = ref [nil[]]
                    in
                        record
                            empty = abs[] = IntStack'empty [Self.get[]] end,
                            push = abs [v] = IntTower'push [Self.get[],v] end,
                            ...
                            MetaObj = MetaSelf.get[]
                        end
                    end
            in
                Self.set[NewInstance];
                Self.get[]
            end
        ...
    in
        ...
    end
```

Figure 8: Source code of seventh model.

metaobjects have an equal interface type, too. As already mentioned on Page 15, methods defined for `IntStack` can be used by other objects if their implementation type is a subtype of `IntStackImpl`. Hence it is necessary to define the implementation types in a way that `IntTowerImpl` is a subtype of `IntStackImpl`. This can be achieved by slightly modifying the implementation type introduced in Figure 7: the object's metaobject information is not stored in a reference cell any more, but as a constant. This is necessary because if type `A` is a subtype of `B`, then `Ref A` is not a subtype of `Ref B` any more.

The implementation of `IntTower` is shown in Figure 8. It is very similar to the one of `IntStack` in the previous section (Figure 7): instance methods are defined in the file scope, instance variables are implemented using reference cells. One may notice that the private method `CanPush`[7] is not part of the implementation type of `IntTower`, but is still has to be defined in the file scope (and not in the declaration part of `IntTower`). A small problem arises when a subclass redefines a method, but still needs to call the inherited method (`IntTower` redefines `push`, but still needs the version implemented for `IntStack`). Since PICT does not support operator overloading [CW85], name mangling is used to unambiguously name methods with the "same" name, but implemented for different classes (refer also to Page 15).

Whenever inheritance is introduced, the question is whether it is possible to redefine

[7] For the complete implementation of `CanPush` refer to Figure 10.

```
val StackClass : ![:T:][!(Stack T)] =
    let
        val pushed = ref[0]
    in
        abs [:T:][] =
            let
                val localPushed = ref[0]
                val Contents = ref [nil[]]
            in
                record
                    empty = abs[] = null[Contents.get[]] end,
                    ...
                    localPushed = abs [] = localPushed.get[] end
                end
            end
        end
    end
```

Figure 9: Source code of generic procedural stack class.

inherited features in a *covariant* or *contravariant* way [Mey88], and what the consequences are. Well, if a method is redefined in a covariant way (e.g., the type of a parameter is replaced by a subtype), the subclass is not a subtype of the superclass any more. Hence, none of the methods or functions defined for the superclass can be used; they have to be all rewritten. By redefining a method in a contravariant way (e.g., replacing the type of a parameter by a supertype), the subclass is still a subtype of the superclass, but the inherited version of a redefined method cannot be used any more.

In principle, we can now model all the features introduced in our object model in section 2, but is there a better solution? We answer this question in the next section.

3.9 Model 8

In this section we will introduce a different approach for modelling objects. It is based on a Scheme [Dyb87] implementation described in [FWH92] (see also Appendix B). The main difference to the previous models is that the stack class is not defined as a *process* abstraction (using `def`), but as a *value* expression (Figure 9). This guarantees that only one metaobject instance per class exists. Instance methods are again defined in the scope of the value expression, and are therefore only accessible by objects of the corresponding class. The implementation described in Figure 9 does not support class methods nor self-reference, but again we can model self-reference using a reference cell according to Model 5 (which is done for the `TowerClass` implementation in Figure 10). Class methods will be reintroduced in the next section.

The careful reader may have noticed that we have mentioned a few times possible extensions to our implementation in order to define generic classes, even in the presence of metaobjects. In order to show an example how generic classes can be defined in PICT, we have slightly changed our implementation and defined a generic stack class `StackClass`. The type of the value `StackClass` is equal to `![:T:][!(Stack T)]`, where the type `Stack`

```
val TowerClass =
    abs [:T:][greater: ![T, T, !Bool]] =
        let
            val Self = emptyRef [:(TowerImpl T):][]
            val Super = StackClass [:T:][]

            def CanPush [v] =                    {- private method -}
                    if ((Self.get[]).empty[]) then
                        true
                    else
                        if (greater [(Self.get[]).top[], v])
                            then true
                            else false
                        end
                    end

            val NewInstance =
                Super with
                    push = abs [v] =
                            if (CanPush[v]) then
                                Super.push[v]
                            else
                                raise[exitOnExn, "Tower: Invalid Push!"]
                            end end
                end
        in
            Self.set [NewInstance];
            Self.get[]
        end
    end
```

Figure 10: Source code of generic procedural tower class.

is defined as

```
type Stack T =
    Record
        push: ![T,Sig],
        pop: ![Sig],
        top: ![!T],
        empty: ![!Bool],
        localPushed: ![!Int]
    end
```

Due to the fact that we have only introduced explicit type annotations where it was needed, only very few changes have to be made to the `IntStack` implementation in order to obtain a generic stack.

To model inheritance, we also take a different approach: we use *delegation semantics*. A subclass does not inherit the features of its superclass, but has an instance of its superclass (Figure 10). A similar approach is used in Self [US87] and Sina [Aks89, Ber94].

Probably this modelling needs a few words of explanation. The `with` operator cannot only be used to extend a `Record` type with new fields (refer to Page 12), but also to extend and redefine a `record` value. If a field exists more than once, only the last field is valid (all others get overwritten). This mechanism is used in `TowerClass` to inherit and redefine features from `StackClass`. Due to the fact that a subclass does not inherit from its superclass, but has an instance instead, only the exported features of the superclass can be accessed; internal representation and implementation is not visible. This has the consequence that dynamic binding is not possible. If the subclass redefines a method which is called by another method defined in an ancestor class (and not redefined in the subclass), the original and not the redefined method will be called. Although this does not seem to be desirable, it could be a first approach for an object model for software composition, because the behaviour of an object does not implicitly depend on its environment; it can only be changed explicitly. In order to achieve dynamic binding, the superclass object must be explicitly told which method implementation has to be called (inherited version or redefined version), or a different approach to implement methods has to be found.

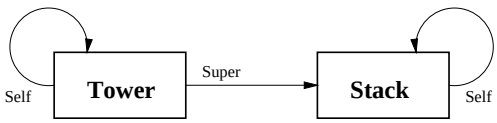
But this model has also its advantages. Since a subclass cannot access the private instance variables of its ancestor, its implementation does not depend on the ancestors implementation. If concurrent objects are defined, we think that we can omit *inheritance anomalies* [MY93] by using our kind of model, but this still is a subject of current research. Another point is that it is not necessary any more to use name mangling for naming methods; a method `foo` is simply called by `(Self.get[]).foo` or `Super.foo`.

Like the stack implementation, the tower implementation described in Figure 10 is generic. The method `CanPush` has to compare the top element with each item to be pushed onto the tower. In the previous model, the operator `>>` is used for that, which is only defined for integers (and its subtypes), but not for any other type. Because PICT does not allow operator overloading[8], each tower object needs a comparison operator valid for the type of elements stored. In order to implement a generic tower, each tower object has to be furnished with the valid comparison operator (for example with a parameter at instantiation like in Figure 10).

3.10 Model 9

In this section, we extend the last model in order to obtain dynamic binding again. What is the problem? When we use inheritance, we instantiate an object of the superclass. When we redefine a method of the superclass, the redefined method will only be used in the subtype. It is not possible to force the superclass object to use the redefined method.

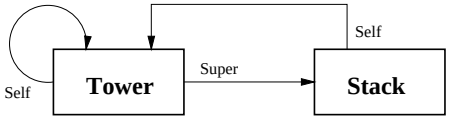
The reason why dynamic binding is not supported by this approach is that `Self` within the instance of the superclass refers to the superclass object, but not to the subclass object:



To achieve the effect of dynamic binding, `Self` of the superclass object has to refer to the

[8] In C++, the operator `>>` would simply be overloaded.

subclass object:



What we need is an instance of a superclass object where `Self` refers to the subclass object. To do so, we introduce so-called *intermediate objects* where all methods and instance variables of a class are defined, but where `Self` is unbound: all methods have an additional first parameter `Self`. The metaobject of each class defines a process `CreateIntermediate` (comparable with a generator in [Coo89, CP94]) where the intermediate object of the class is defined:

```
def CreateIntermediate[] =                {- defined in IntStackClass -}
    let
        val LocalPushed = ref[0]          {- private instance variables -}
        val Contents = ref[nil[]]
    in
        record
            empty = abs [Self: IntStackImpl] = null[Contents.get[]] end,
            push = abs [Self: IntStackImpl, v] = ... end,
            ...
            localPushed = abs [Self: IntStackImpl] = LocalPushed.get[] end
        end
     end
```

In the `Create` method of the metaobject, an intermediate object is created, each exported method is bound to a method defined in the intermediate object, and the correct binding of `Self` is established. Like in the previous model, an empty reference cell is used to model self-reference.

```
def Create[] : IntStack =              {- defined in IntStackClass -}
    let
        val IntStackIntermediate = CreateIntermediate[]
        val Self = emptyRef[:IntStackImpl:][]
        val NewInstance =
            record
                empty = abs [] = IntStackIntermediate.empty[Self.get[]] end,
                push = abs [v] = IntStackIntermediate.push[Self.get[], v] end,
                ...
            end
     in
         Self.set[NewInstance]
         Self.get[]
     end
```

Besides the implementation of all public class methods and the method `Create`, which creates new objects (like in the previous model), the metaobject exports the method `CreateIntermediate`. This method returns a fresh copy of an intermediate object of the class.

The modelling of inheritance is now straightforward. In order to reuse the methods defined in an ancestor class, the metaobject of a class gets a fresh copy of the intermediate object

of its direct superclass. This intermediate object is then used to define the intermediate object of the class itself. It is possible to (i) override methods, (ii) define new methods, and (iii) call inherited methods.

```
def CreateIntermediate[] =              {- defined in IntTowerClass -}
    let
        val SuperIntermediate = IntStackClass.CreateIntermediate[]
        def CanPush [Self: IntTowerImpl, v] = ...
                {- check whether v is smaller than top -}
    in
        SuperIntermediate with
            push = abs [Self: IntTowerImpl, v] =
                    if (CanPush[Self,v]) then
                        SuperIntermediate.push[Self,v]
                    else
                        raise[exitOnExn,"Tower : Invalid Push"]
                    end end
        end
    end
```

The `Create` method of the tower metaobject is defined in the same way as in the stack class. For the complete code of the stack and tower classes, refer to Appendix C.

A further change in comparison to Model 4 is that private instance variables have been removed from the implementation type. The way we use delegation does only allow the access to public features. We think that the implementation type can be used to model *protected* features, but this is a topic of future research.

The reader should note that this model has the disadvantage that the method `CreateIntermediate` is exported by the metaobject. This violates data encapsulation, but it is necessary to obtain code reuse. There is no need that a client of an object uses this method.

4 Conclusion and future work

In the last section we have introduced several implementations of object models. The idea behind the work was not to find only one implementation for a specific model, but to obtain an object modelling workbench for PICT. Using this workbench, it will be easier to define a suitable object model for software composition. We did not only try to keep the implementations simple, but also tried to write generic code in order to reuse code as much as possible. All the models we have presented so far are sequential models. Although the goal is to find a concurrent object model, we have concentrated on sequential aspects of object models as a first step. Besides finding other implementations, the definition and mapping of a concurrent object model will be the goal of our future work, as a next step towards an object model for a composition language.

PICT is a suitable language for modelling objects, although there are a few language features which make modelling a bit difficult. First of all, recursive types do not only require explicit folding and unfolding by the programmer, but no subtype relation can be established between them. Reference cells are very handy to model variables, but they do

require a more sophisticated concurrency control, and they interfere with subtype relations (if the type `A` is a subtype of `B`, then `Ref A` is neither a subtype of `Ref B` nor vice versa), too.

There are some features which could be introduced into PICT in order to obtain a more efficient code reuse. For example, it should be possible to annotate the type parameters of a type operator with an upper bound[9], like

```
type T = ...
type T0 (X: T) = ...
```

The type operator `T0` can now only be used when its type parameter is of type `T`. Especially in the context of defining record types for objects this would be very handy. A useful extension would be a type operator to merge two given `Record` types to get one record type, and another operator to merge two record values to form one record.

In PICT it is possible to have anonymous process definitions (using `abs`), but it is not possible to have anonymous type definitions. Sometimes it would be easier to declare a value (with no type annotation) and use the type of the previously declared value to define another type or value. This could be achieved by introducing a `typeof` operator, returning the static type of an expression. This operator could then be used in a context where a type is required.

Last but not least, in none of our object models it was possible to compare objects. Since objects are modelled as records, a generic comparison operator for records is required, which is not available in the current version of PICT. A solution to this problem is the assignment of an unique integer value to each object, which can be used to compare objects. Of course this can only be used to check whether to variables have a reference to the same object, but it is not possible to compare to nonidentical objects having the same abstract representation.

Probably a very challenging work would be to find a minimal set of extensions to PICT in order to get a real object-oriented programming language (like CLOS is an object-oriented extension to Common Lisp [KdRB91]).

Acknowledgements

We thank all the members of the Software Composition Group for their support of this work, especially Oscar Nierstrasz and Patrick Varone, and Benjamin Pierce and Manuel Barrio for reviewing and many helpful comments.

References

[Aks89] Memeth Aksit. *On the Desing of the Object-Oriented Language Sina.* PhD thesis, University of Twente, NL, 1989.

[9] Similar to constraint genericity in Eiffel.

[Ber94] Lodewijk Bergmans. *Composing Concurrent Objects.* PhD thesis, University of Twente, NL, June 1994.

[BF95] A. Burda and G. Färber. *Das grosse Buch zu DELPHI.* Data Becker, 1995.

[BL94] Michel Beaudouin-Lafon. *Object-oriented Languages: Basic principles and programming techniques.* Chapman & Hall, 1994.

[Bor91] Open Architecture Handbook. Borland International Inc., Scotts Valley, CA, 1991.

[BS95a] Manuel Barrio Solorzano. *Estudio de Aspectos Dinamicos en Sistemas Orientados al Objecto.* PhD thesis, Universidad de Valladolid, September 1995.

[BS95b] Søren Brandt and René W. Schmidt. The Design of a Meta-Level Architecture for the BETA Language. In *Proceedings of META '95: Workshop on Advances in Metaobject Protocols and Reflection at ECOOP '95*, August 1995.

[Chi95] Shigru Chiba. A Metaobject Protocol for C++. In *Proceedings of OOPSLA '95*, volume 30 of *ACM SIGPLAN Notices*, pages 285–299, October 1995.

[Coo89] William R. Cook. *A Denotational Semantics of Inheritance.* PhD thesis, Department of Computer Science, Brown University, Providence, RI, May 1989.

[CP94] William Cook and Jens Palsberg. A denotaional semantics of inheritance and its correctness. *Information and Computation*, 114(2):329–350, 1994.

[CW85] Luca Cardelli and Peter Wegner. On Understanding Types, Data Abstraction, and Polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dyb87] R. Kent Dybvig. *The SCHEME Programming Language.* Prentice Hall, 1987.

[FWH92] Daniel P. Friedman, Mitchell Wand, and Christopher T. Hayens. *Essentials of Programming Languages.* McGraw-Hill, 1992.

[GR89] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison-Wesley, September 1989.

[Jon93] Cliff B. Jones. A Pi-Calculus Semantics for an Object-Based Design Notation. In E. Best, editor, *Proceedings of CONCUR'93*, LNCS 715, pages 158–172. Springer, 1993.

[KdRB91] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[Lie86] Henry Lieberman. Using Prototypical Objects to Implement Shared Behaviour in Object Oriented Systems. In *Proceedings OOPSLA '86*, volume 21 of *ACM SIGPLAN Notices*, pages 214–223, November 1986.

[Men94] Tom Mens. A survey on formal models for OO. Technical Report vub-tinf-tr-94-03, Department of Computer Science, Vrije Universiteit Brussel, Belgium, 1994.

[Mey88] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[Mey92] Bertrand Meyer. *Eiffel: the Language*. Prentice Hall, 1992.

[Mil89] Robin Milner. A calculus of mobile processes, part I+II. Technical Report ECS-LFCS-89-85, Computer Science Department, University of Edinburgh, UK, 1989.

[Mil91] Robin Milner. The polyadic Pi-calculus: a tutorial. Technical Report ECS-LFCS-91-180, Computer Science Department, University of Edinburgh, UK, October 1991.

[MY93] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In Gul Agha, Peter Wegner, and Akinori Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[ND95] Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3–28. Prentice Hall, 1995.

[NGT92] Oscar Nierstrasz, Simon Gibbs, and Dennis Tsichritzis. Component-Oriented Software Development. *Communications of the ACM*, 35(9):160–165, September 1992.

[Nie92] Oscar Nierstrasz. Towards an Object Calculus. In Mario Tokoro, Oscar Nierstrasz, and Peter Wegner, editors, *Proceedings of ECOOP 1991 Workshop on Object-based Concurrent Computing*, LNCS 612, pages 1–20. Springer, 1992.

[NM95] Oscar Nierstrasz and Theo Dirk Meijler. Requirements for a Composition Language. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, LNCS 924, pages 147–161. Springer, 1995.

[Pae93] Andreas Paepcke, editor. *Object-Oriented Programming: The CLOS Perspective*. MIT Press, 1993.

[Pie95a] Benjamin C. Pierce. *Pict User Manual*. Computer Laboratory, University of Cambridge, UK, May 1995.

[Pie95b] Benjamin C. Pierce. Programming in the Pi-Calculus: An experiment in concurrent language design. Technical report, Computer Laboratory, University of Cambridge, UK, May 1995. Tutorial Notes for Pict Version 3.6a.

[PT95] Benjamin C. Pierce and David N. Turner. Concurrent Objects in a Process Calculus. In Takayasu Ito and Akinori Yonezawa, editors, *Theory and Practice of Parallel Programming (TPPP)*, LNCS 907, pages 187–215. Springer, April 1995.

[Str94] Bjarne Stroustrup. *The Design and Evolution of C++*. Addison-Wesley, 1994.

[Tur96] David N. Turner. *The Polymorphic Pi-Calculus: Theory and Implementation.* PhD thesis, Department of Computer Science, University of Edinburgh, UK, 1996.

[US87] David Ungar and Randall B. Smith. SELF: The Power of Simplicity. In *Proceedings OOPSLA'87*, volume 22 of *ACM SIGPLAN Notices*, pages 227–242, December 1987.

[Var96] Patrick Varone. Implementation of "Generic Synchronization Policies" in PICT. Technical Report IAM-96-005, University of Bern, Institute of Computer Science and Applied Mathematics, February 1996.

[Wal95] David J. Walker. Objects in the Pi-Calculus. *Information and Computation*, 116(2):253–271, 1995.

A Complete IntStack and IntTower classes

```
Stack = class {
        fields
                stack: array[1..N] of integer;
                top: integer;
        methods
                procedure Push (value: integer);
                procedure Pop ( );
                function Top ( ): integer;
}

procedure Stack.Push (value: integer) {
        top := top + 1;
        stack[top] := value;
}

procedure Stack.Pop ( ) {
        top := top - 1;
}

function Stack.Top ( ): integer {
        return stack[top];
}

Tower = class Stack {
        methods
                procedure Push (value: integer);
                function CanPush (value: integer): boolean;
}
```

```
function Tower.CanPush (value: integer): boolean {
      if top = 0
            then return true;
            else return value < Top ( );
}

procedure Tower.Push (value: integer) {
      if CanPush (value)
            then Stack.Push (value);
            else error.Write ("push impossible");
}
```

B Procedural stack class implementation in Scheme

This Scheme implementation can be found in [FWH92] on page 215.

```
(define make-stack
   (let ((pushed 0))
      (lambda ()
         (let ((stk '()) (local-pushed 0))
            (lambda (message)
               (case message
                  ((empty?) (lambda () (null? stk)))
                  ((push!) (lambda (x)
                        (set! pushed (+ pushed 1))
                        (set! local-pushed (+ local-pushed 1))
                        (set! stk (cons x stk))))
                  ((pop!) (lambda ()
                        (if (null? stk)
                           (error "Stack: Underflow")
                           (begin
                              (set! pushed (- pushed 1))
                              (set! local-pushed (- local-pushed 1))
                              (set! stk (cdr x stk))))))
                  ((top) (lambda ()
                        (if (null? stk)
                           (error "Stack: Underflow")
                           (car stk))))
                  ((local-pushed) (lambda () local-pushed))
                  ((pushed) (lambda () pushed))
                  (else (error "Stack: Invalid message" message))))))))
```

C Procedural stack classes with dynamic binding

```
{---- class IntStack ------------------------------------------------------}

type IntStack =
    Record
        empty: ![!Bool],
        push: ![Int,Sig],
        pop: ![Sig],
        top: ![!Int],
        localPushed: ![!Int]
end

type IntStackImpl = IntStack

type IntStackIntermediate =
    Record
        empty: ![IntStackImpl,!Bool],
        push: ![IntStackImpl,Int,Sig],
        pop: ![IntStackImpl,Sig],
        top: ![IntStackImpl,!Int],
        localPushed: ![IntStackImpl,!Int]
end

type IntStackClass =
    Record
        pushed: ![!Int],
        GetClassName: ![!String],
        Create: ![!IntStack],
        CreateIntermediate: ![!IntStackIntermediate]
end

val IntStackClass =
    let
        val MetaSelf = emptyRef [:IntStackClass:][]  {- Self reference -}
        val Pushed = ref[0]                          {- class variable -}

        def pushed [] = Pushed.deref[]               {- class method -}

        def CreateIntermediate [] : IntStackIntermediate =
            let
                val LocalPushed = ref[0]
                val Contents = ref[nil[]]

                def empty [Self: IntStackImpl] = null[Contents.deref[]]

                def push [Self: IntStackImpl, newval] =
                        LocalPushed.set[LocalPushed.deref[] + 1];
                        Pushed.set[Pushed.deref[] + 1];
                        Contents.set[cons [newval, Contents.deref[]]]

                def pop [Self: IntStackImpl] =
                        if (Self.empty[]) then
                            raise[exitOnExn,"Pop on empty stack!"]
                        else
                            Pushed.set[Pushed.deref[] - 1];
                            LocalPushed.set[LocalPushed.deref[] - 1];
```

```
                        Contents.set[cdr[Contents.deref[]]]
                    end

                def top [Self: IntStackImpl] =
                        if (Self.empty[]) then
                           raise[exitOnExn,"Top on empty stack!"]
                        else
                            car[Contents.deref[]]
                        end

                def localPushed [Self: IntStackImpl] = LocalPushed.deref[]

            in
                record
                    empty = empty,
                    push = push,
                    pop = pop,
                    top = top,
                    localPushed = localPushed
                end
            end

        def Create [] : IntStackImpl =
            let
                val Self = emptyRef [:IntStackImpl:][]     {- Self reference -}
                val IntermediateSelf = CreateIntermediate[]

                val NewInstance =
                    record
                        empty = abs[] = IntermediateSelf.empty[Self.deref[]] end,
                        push = abs[v] = IntermediateSelf.push[Self.deref[],v] end,
                        pop = abs[] = IntermediateSelf.pop[Self.deref[]] end,
                        top = abs[] = IntermediateSelf.top[Self.deref[]] end,
                        localPushed = abs[] =
                                IntermediateSelf.localPushed[Self.deref[]] end
                    end
            in
                Self.set[NewInstance];                    {- binding of Self -}
                Self.deref[]
            end

        val NewMetaInstance =
            record
pushed = pushed,
                GetClassName = abs [] = "IntStackClass" end,
                Create = Create,
                CreateIntermediate = CreateIntermediate
            end
    in
        MetaSelf.set[NewMetaInstance];                    {- binding of Self -}
        MetaSelf.deref[]
    end

{---- class IntTower ------------------------------------------------------}

type IntTower = IntStack
```

```
type IntTowerImpl = IntTower

type IntTowerIntermediate = IntStackIntermediate
    with
        push: ![IntTowerImpl,Int,Sig]
end

type IntTowerClass = IntStackClass
    with
        Create: ![!IntTower],
        CreateIntermediate: ![!IntTowerIntermediate]
end

val IntTowerClass =
    let
        val MetaSelf = emptyRef [:IntTowerClass:][]  {- Self reference -}

def CreateIntermediate [] =
            let
                val SuperIntermediate = IntStackClass.CreateIntermediate[]

                def CanPush [Self: IntTowerImpl, v] =
                        if (Self.empty[]) then
                           true
                        else
                            if (Self.top[] >> v)
                                then true
                                else false
                            end
                        end

                def push [Self: IntTowerImpl, newval] =
                        if (CanPush[Self,newval]) then
                            SuperIntermediate.push[Self,newval]
                        else
                            raise[exitOnExn,"Tower: Invalid Push!"]
                        end

            in
                SuperIntermediate
                    with
                        push = push
                end
            end

        def Create[] : IntTowerImpl =
            let
                val Self = emptyRef [:IntTowerImpl:][]     {- Self reference -}
                val IntermediateSelf = CreateIntermediate[]

                val NewInstance =
                    record
                        empty = abs[] = IntermediateSelf.empty[Self.deref[]] end,
                        push = abs[v] = IntermediateSelf.push[Self.deref[],v] end,
                        pop = abs[] = IntermediateSelf.pop[Self.deref[]] end,
                        top = abs[] = IntermediateSelf.top[Self.deref[]] end,
                        localPushed = abs[] =
```

```
                            IntermediateSelf.localPushed[Self.deref[]] end
            end
        in
            Self.set[NewInstance];
            Self.deref[]
        end

    val NewMetaInstance = IntStackClass
        with
            GetClassName = abs[] = "IntTowerClass" end,
            Create = Create,
            CreateIntermediate = CreateIntermediate
        end
in
    MetaSelf.set[NewMetaInstance];
    MetaSelf.deref[]
end
```