Building Shelf Systems

A Case Study in Structured ML Programming

Peter Padawitz
Fachbereich Informatik
Universität Dortmund, Germany
peter@ls5.informatik.uni-dortmund.de

Forschungsbericht
Nr. 521/1994

Abstract

We present a generic functional program for loading (two-dimensional) containers with arbitrary objects, filling shelves with items of different kinds and - last not least - designing complete shelf systems. This piece of software and its development is a case study in functional programming for solving hierarchical configuration problems. Such problems require tailor-made, but nevertheless abstract and generic data models and algorithms. We claim that these are more directly realizable in a functional language than in a classical imperative language. Polymorphism, module abstraction, semantically well-founded exception handling and the problem-oriented option for static data, "lazy" structures (streams) or dynamic objects are the main concepts of functional programming we are going to use and explain in the context of our case study.

Contents

1 Introduction **2**

2 Objects and Containers **4**

3 The Loading Strategy **6**

4 ASCII Output **10**

5 Instances of *Container* and *Display* **12**

6 A Design Session **16**

7 Streams, Traversals and String Primitives **20**

8 PostScript Output **23**

9 Conclusion **26**

10 Acknowledgement **27**

1 Introduction

A number of functional programming languages (cf. e.g. [Dav 92], [Pau 91], [Ull 94], [Tur 90]) are under development and use in different application areas. For the various motivations to propagate these languages, cf. e.g. [BW 88] and [Rea 89]. In contrast to the well-known imperative or procedural languages and the associated programming style, functional languages admit rather abstract formulations of algorithmic problem solutions, which considerably simplify the validation of a program and the comparison of several solutions against each other. Even if a functional language is not chosen for the final implementation, it provides a formal framework for *specifying* problem solutions at high levels in an overall design process. Moreover, functional languages are useful for the rapid prototyping of algorithmic problem solutions because they combine a simple, but precise, algebraic semantics (cf. [Pad 92]) with operational features for controlling the program behaviour at runtime. Many specification languages (cf. e.g. [GH 93], [Pad 94]) and their "proof assistants", which support the design process, adopt the functional view insofar as their roots lie in algebra and term rewriting (cf. [DJ 90], [Wir 90]).

This paper presents a case study in solving a configuration problem by using functional programming concepts. We want to

- load containers with arbitrary objects,
- fill shelves with books and HiFi components with respect to certain constraints, and
- build complete shelf systems

in a highly structured way. Similarities of as well as differences between these tasks will be reflected in the structure of the program, which is written in *Standard ML* (cf. [Pau 91], [Ull 94]). ML provides the concepts of polymorphism for single functions and functoriality for modules. Indeed these are the main means, which allow us to approach the three tasks at the same time. *Functors* are ML's parameterized modules, while *structures* and substructures realize the *inheritance* known from object-oriented programming. The main concept of object-oriented programming, namely the creation of dynamic objects via instantiating *classes*, is also available in ML. In fact, a class in ML is a nullary functor including a dynamic object, which is initialized by each call of the functor (cf. [MQu 84], [Pad 93]).

Two functors, *Container* and *Display*, mainly provide the filling algorithm and a procedure for visualizing loadings (cf. Sections 2 and 4). Then three classes, *CONTAINER, SHELF* and *SHELFSYSTEM*, involve three different calls of *Container* and *Display* and thus become heirs of these functors (cf. Section 5). *CONTAINER* realizes the task of loading arbitrary containers with arbitrary objects. *SHELF* is specialized to filling shelves with books and HiFi components. *SHELFSYSTEM* provides the entities for building and displaying shelf systems. The objects of *SHELFSYSTEM* are the containers of *SHELF*. This is reflected in the program insofar as the structure *SH*, which defines the objects of *SHELFSYSTEM*, is an object of the class *SHELF*.

The functors *Container* and *Display* keep to static, immutable data. Only *CONTAINER, SHELF* and *SHELFSYSTEM* are defined as classes for the purpose of handling dynamic objects. Here these objects are streams of permutations, from which, at any stage of the shelf design process, the loading procedure takes the first element and regards it as the order in which the given objects are to be packed. If this is not possible, the loading procedure changes the order gradually. Since the original order may considerably affect the search time for a fitting permutation, the user can interactively select a new order and restart the loading algorithm (cf. Sections 5 and 6).

The first goal of this case study is to propagate structured functional programming. The second one is to integrate functional and object-oriented concepts. For achieving the latter we need not define a new language, but we need examples, which serve as prototypes for a reasonable separation of naturally functional from naturally object-oriented issues. We come back to this point when drawing conclusions from the case study (cf. Section 9).

We assume a little familiarity with ML (cf. e.g. [Pau 91], [Ull 94]) when stepwise presenting and explaining the program in the following four sections.

2 Objects and Containers

The functor *Container* takes a structure parameter *Obj* of the following signature *Object.*

```
signature Object  =  sig  eqtype obj
                          val le : obj -> int
                          val he : obj -> int
                          val topConstraint : obj -> bool
                          val Aligned : bool
                          val Symmetric : bool
                     end
```

The signature assumes that each object to be loaded comes with a type (*obj*) and two integer values denoting the length (*le*) resp. height (*he*) of the object. Each two objects must be comparable via an equality relation *eq* and each object must be equipped with a top constraint, which determines when it may be put on top of another object. Finally, Boolean values *Aligned* and *Symmetric* indicate whether or not all objects are to be aligned or piled up symmetrically. We proceed with the details of *Container*:

```
functor    Container(structure Obj :  Object) = struct open Obj Aux[1]
```

The following abstract datatype *cont* provides the container constructors *new* and *add*. *new(l,h)* denotes an empty container with length *l* and height *h*. *add(a,x,y,c)* represents the container constructed from the container *c* by putting *a* into *c* such that *(x,y)* is the position of the leftmost-lowest corner of *a* in *c*.

```
abstype cont = new of int * int | add of obj * int * int * cont

with    val     New = new
        val     Add = add

        fun     len(new(l,_)) = l
        |       len(add(_,_,_,c)) = len(c)

        fun     hei(new(_,h)) = h
        |       hei(add(_,_,_,c)) = hei(c)

        fun     isNew(new _) = true
        |       isNew(add _) = false

        fun     top(add(a,_,_,_)) = a

        fun     pop(add(_,_,_,c)) = c

        fun     Objs(new _) = nil
        |       Objs(add(a,_,_,c)) = a::Objs(c)

        fun     pos(add(a,x,y,c),b) = if a = b then (x,y) else pos(c,b)

        exception undef
```

[1] cf. Section 7

```
        fun     get(new _,_,_) = raise undef
        |       get(add(a,x,y,c),i,j)
                 = if inside(a,x,y)(i,j) then a else get(c,i,j)
        and     inside(a,x:int,y:int)(i,j)
                 = x <= i andalso i < x+le(a) andalso y <= j andalso j < y+he(a)

        fun     frame(new(l,h)) = new(l+2,h+2)
        |       frame(add(a,x,y,c)) = add(a,x+1,y+1,frame(c))

end (* cont *)
```

len(c) and *hei(c)* denote the length and height of *c*, respectively. Further basic operations for building containers and accessing their contents are encapsulated into the abstract datatype definition, which hides the constructors *new* and *add* from all functions except from those defined between *with* and *end*. On the one hand, *cont* yields the interface for refinements of *Container* insofar as only the functions in the capsule need to be redefined when containers are actually implemented. On the other hand, all functions defined outside the capsule are independent of the actual implementation.

Abstract data type definitions considerably reduce the verification task put forth by an implementation of the data type: The entire implementation is correct if and only if each function of the `abstype` is equivalent to its counterpart in the implemented datatype. Since *initial semantics* provides the model- and proof-theoretic basis for most ML programs (cf. [Pad 92]), one may employ a suitable proof assistant like *Expander* (cf. [Pad 94]) for proving such equivalences.

```
fun     free(c,x,y) = (get(c,x,y);false) handle undef => true

val     occupied = not o free

fun     onTop(c,x,1) = true
|       onTop(c,x,y) = occupied(c,x,y-1)

fun     aligned(c,a,x,1) = true
|       aligned(c,a,x,y) = occupied(c,x,y-1) andalso
                           let val b = get(c,x,y-1)
                               val (i,_) = pos(c,b)
                           in x = i andalso le(a) = le(b) end

fun     symmetric(c,a,x,1) = true
|       symmetric(c,a,x,y) = occupied(c,x,y-1) andalso
                             let val b = get(c,x,y-1)
                                 val (i,_) = pos(c,b)
                                 val j = le(a)+x-i
                             in x <= i andalso j <= le(b)
                                       andalso le(b) <= j+1 end
```

Some of these predicates are checked before a new object is put into a container. Since they are defined outside *cont*, none of them needs to be redefined by an implementation of *cont* (see above).

3 The Loading Strategy

For the actual design part of the program we must select a concrete strategy for loading containers. In which order is a given set of objects to be loaded into a given set of empty containers? Defining such a strategy involves setting exit points where the execution of a substrategy is stopped and the control is passed over onto a higher level of execution. In ML, an exit is implemented by *raising* an exception. Resuming the control at the higher level corresponds to *handling* the exception. According to the different functions, which may be called by the overall loading function *fill*, we provide the following exceptions:

```
exception   noFilling and nextPerm of obj list * cont list
exception   Full and Restart of int
```

While *noFilling* and *Full* have no parameters, *nextPerm* and *Restart* have arguments, which transfer certain information from point where the exception is raised to the point where it is handled.

The loading procedure starts by calling the function *fill*, which passes the control over to - at the whole - eight auxiliary functions *fillConts, unFill, tryPerms, addSome, tryAdd, AddOrRestart, onTop* and *Free*. The first three have mutually recursive definitions, while the other five comprise a calling hierarchy, which proceeds from loading several objects (*addSome*) via searching for a position for the next object to be entered (*tryAdd*) to checking the constraints to be satisfied if a selected object is put at a given position (*AddOrRestart*) and, on the lowest level, to traversing single points of the rectangle to be occupied by the selected object (*onTop* and *Free*). Let us start with the main function *fill*:

```
val     Sizes = map(fn(a)=>(le(a),he(a)))
val     newConts = map(fn(l,h)=>New(l,h))

fun     fill(objs)((l,h)::sizes)
          = if (sum_of_prods o Sizes)(objs) <= l*h+sum_of_prods(sizes)
            then fillConts(objs)[New(l,h)](newConts(sizes))
                 handle nextPerm(objs,full) => unFill(objs)(full)(nil)
            else raise noFilling
```

fill tries to enter the objects listed in *objs* into empty containers whose lengths and heights are listed in *sizes*. If the sum of *sizes* exceeds the sum of the sizes of the objects, *objs* does not fit into the containers and the exception *noFilling* is raised and passed over to the caller of *fill*.

```
and     fillConts(nil)(conts) _ = conts
|       fillConts(objs)(c::full)(empty)
          = let val (c,rest) = addSome(objs)(c)
                val conts = c::full
            in if rest = nil then conts
               else if null(empty) then raise nextPerm(rest,conts)
                    else fillConts(rest)(hd(empty)::conts)(tl(empty)) end

and     unFill _ (nil) _ = raise noFilling
|       unFill(objs)(c::full)(empty)
          = if isNew(c) then unFill(objs)(full)(c::empty)
            else let val objs = top(c)::objs
                     val conts = pop(c)::full
                 in (tryPerms o tail o permutations)(objs)(conts)(empty)
                    handle noFilling => unFill(objs)(conts)(empty) end
```

Compared to *fill, fillConts* and *unFill* have a further parameter: the list *conts* = *c::full* of containers where *c* is the container to be filled next and whose *full* consists of all containers filled up completely. *fill* calls *fillConts* with *conts* consisting of the empty containers constructed from *sizes.* If *objs* does not fit in the given order into *conts*, then *fillConts* raises the exception *nextPerm* with the list of unpacked objects and the list of nonempty containers. Later *fill* handles *nextPerm* by calling *unFill* with those parameters and an empty list of empty containers. *unFill* unloads the object loaded at latest, adds it to the objects still to be packed and calls *tryPerms*:

```
and     tryPerms(a)(mt) _ _ = raise noFilling
|       tryPerms(a)(perm&perms)(conts)(empty)
          = if a = hd(perm) then tryPerms(a)(perms())(conts)(empty)
            else fillConts(perm)(conts)(empty)
                 handle nextPerm _ => tryPerms(a)(perms())(conts)(empty)
```

tryPerms computes a new permutation[2] *perm* of the extended object list and calls *fillConts*, which tries to enter the objects into *conts* in the order given by *perm.* Permutations that start with the object *a*, which was unpacked by *unFill*, the caller of *tryPerms*, are skipped (see below).

Even in the new order *objs* may not fit into *conts.* Then, again, *fillConts* raises *nextPerm.* But since *fillConts* has been called by *tryPerms* and not by *fill* as above, it is also *tryPerms*, which handles *nextPerm.* In fact, *tryPerms* handles *nextPerm* not in the way *fill* does. While *fill* needed the parameters of *nextPerm, tryPerms* does not. In other words, the recursive call of *tryPerms* does not depend on the "loading state" in which *nextPerm* was raised. *tryPerms* receives the next permutation of *objs*, but the same lists *conts* and *empty* it obtained in the first call.

If no permutation of *objs* fits into *conts* and *empty*, then *tryPerms* raises the exception *noFilling.* Since *unFill* was the first caller of *tryPerms*, control is passed to *unFill*, which handles *noFilling* by a recursive call. A further object is unloaded and added to the objects to be packed and permuted. To sum up, each call of *unFill* extends the list *objs* of unpacked objects by the object *a*, which was packed at latest, and creates a loop over all permutations of *a::objs.* Since *a* is unpacked only if no permutation of *objs* fits, each permutation of *a::objs*, which begins with *a*, has already been tested previously. Hence *tryPerms* skips such permutations.

The loading process stops if an order has been found in which the objects can be placed into *conts* or if *unFill* accesses only empty containers. In the first case *fillConts* returns *conts* and hands it over to its caller *fill.* In the second case all permutations of the original object list have failed and *unFill* raises the exception *noFilling*, which is then transmitted through all recursive calls of *unFill* to *fill. fill* does not handle *noFilling.* The exception is passed over to the caller of *fill* (cf. Section 5).

Let us now look at the local part of the algorithm, i.e. how a single container is filled up. *fillConts* takes the first element *c* of *conts* and calls *addSome(objs)(c).*

```
and     addSome(nil)(c) = (c,nil)
|       addSome(a::objs)(c) = let val (c,rest) = addSome(objs)(c)
                                  val (l,h) = (len(c),hei(c))
                              in (tryAdd(a,c)(1,1)(1)(l,h),rest)
                                 handle Full => (c,a::rest) end
```

addSome puts as many objects as possible into the empty container *New(l,h)* and returns the pair the filled container *c* and the list *rest* of objects, which do not fit into *c.* For entering a single object *a* into *c*, *addSome* calls

[2]For the generation of permutations, cf. Section 7.

tryAdd. If a does not fit into c, then *tryAdd* raises the exception *Full*, which indicates that c is full. *addSome* handles *Full* by returning an unchanged container c and adding a to the list *objs* of the remaining objects.

```
and     tryAdd(a,c)(i,j)(start)(l,h)
          = AddOrRestart(a,c)(i,j)(l,h)
            handle Restart(i)
                    => if i < l then tryAdd(a,c)(i+1,j)(start)(l,h)
                       else if j < h then tryAdd(a,c)(start,j+1)(start)(l,h)
                            else raise Full
```

tryAdd(a,c) traverses all positions of c between *(i,j)* and *(l,h)* and searches for the first position, which may become the leftmost-lowest corner of a in c. At each of these positions *tryAdd* calls *AddOrRestart* and continues the search only if *AddOrRestart* has raised the exception *Restart*. *Restart* comes with a parameter i, which is the vertical axis of the position *(i,j)* checked at last. If $i < l$, then *(i+1,j)* becomes the next possible start position for a. If $i \geq l$ and $j < h$, then *tryAdd* proceeds with the new start position *(start,j+1)* where *start* is the leftmost vertical axis of c. If $i \geq l$ and $j \geq h$, then there is no position left and *tryAdd* raises the exception *Full.*

```
and     AddOrRestart(a,c)(i,j)(l,h)
         = let val (x,y) = (i+le(a)-1,j+he(a)-1)
           in if y > h then raise Full
              else if x > l then raise Restart(l)
                   else if free(c,i,j) andalso
                           (j = 1 orelse topConstraint(a)) andalso
                           (not(Aligned) orelse aligned(c,a,i,j)) andalso
                           (not(Symmetric) orelse symmetric(c,a,i,j))
                        then (forward(OnTop(c,j))(i)(x); forward(Free(c,j))(i)(x);
                              Add(a,i,j,c))
              else raise Restart(i) end

and     OnTop(c,j)(i) = onTop(c,i,j) orelse raise Restart(i)

and     Free(c,j)(i) = free(c,i,j) orelse let val b = get(c,i,j)
                                              val (x,_) = pos(c,b)
                                          in raise Restart(x+le(b)-1) end

end (* Container *)
```

AddOrRestart first checks whether the remaining free space of c is large enough for a. If the height of c does not suffice, then *AddOrRestart* raises *Full.* Since *Full* is not handled by *tryAdd*, the search for a start position for a stops and *Full* is passed over to *addSome.* If only the length of c does not suffice for a, then *Restart* is raised with the rightmost vertical axis l of c so that *tryAdd* will proceed to the next upper horizontal axis.

If a fits into c, then *AddOrRestart* checks all constraints, which depend on c, a and the current position *(i,j):* Is *(i,j)* free? Does a satisfy the top constraint? In case that *Aligned* is set to true, can the sides of a be aligned to the sides of the object below a? In case that *Symmetric* is set to true, can a be put symmetrically on top of the object below a? If all these conditions are satisfied, *forward*[3] calls the function *OnTop(c,j)* on all integers between i and x and thus checks whether a would lie completely on top of other objects. If this is not

[3]cf. Section 7.

true, then *OnTop(c,j)* raises *Restart* with the leftmost vertical axis $X \geq i$ such that the position *(X,j-1)* is free. Hence *tryAdd* will proceed to the start position *(X+1,j)* (or *(start,j+1)* if $X \geq l$. Analogously, *forward* calls the function *Free(c,j)* on all integers between i and x and thus checks whether all positions between *(i,j)* and *(x,j)* are free. If an object b occupies such a position, then *Free(c,j)* raises *Restart* with the rightmost vertical axis X occupied by b. If all constraints are satisfied, then a is put into c. Otherwise *AddOrRestart* raises *Restart* with the current vertical axis i.

Figure 3.1 depicts the relations between all functions comprising the loading strategy. A circular node represents a function. A rectangular vertex denotes an exception. An arc from a function f to a function g labelled by *calls* indicates that f calls g. An edge from a function f to an exception e marked by *raises* says that f raises g. An arc a with two sources, namely a function f and an exception e, indicates that f handles e by calling the function resp. raising the exception at the target of a. For instance, *fill* handles *nextPerm* by calling *unFill*, while *tryAdd* handles *Restart* by raising *Full* or calling *tryAdd* recursively.

Fig. 3.1. The functional structure of the loading procedure

4 ASCII Output

The functor *Display* takes a structure parameter *Picts* of the following signature *Pictures*.

```
signature Pictures
           = sig eqtype obj
                 type cont
                 val len : cont -> int
                 val hei : cont -> int
                 val get : cont * int * int -> obj
                 val pos : cont * obj -> int * int
                 val free : cont * int * int -> bool
                 val space : int
                 val interior : int * int -> obj -> int * int -> string
                 val corner : obj -> int * int -> string
                 val horiz_edge : obj -> int * int -> string
                 val left_edge : obj -> int * int -> string
                 val right_edge : obj -> int * int -> string
                 val column : obj -> int * int -> string
             end
```

Each string-valued function of *Pictures* is to determine the representation of a specific part of an object. *interior* says how the inside part should look like. *horiz_edge* and *corner* represent parts of horizontal edges. *left_edge (right_edge)* depicts the left (right) vertical edges. *column* visualize those objects, which cover a single horizontal axis.

Display starts by opening the parameter *Picts*, which means that all the above functions become available to *Display*.

```
functor Display(structure Picts : Pictures) = struct open Picts

fun     container(c,file) = let val file = open_out(file)
                                 val (l,h) = (len(c),hei(c))
                             in Aux.ruler(picture(c))(space,file,l,h);
                                close_out(file) end
```

The function *picture* takes a container *c* and a point *(i,j)* of the plane representing *c* and assigns to *(i,j)* the string function, which corresponds to the relation between *(i,j)* and the object occupying this position.

```
and     picture(c)(i,j)
         = if free(c,i,j) then Aux.blanks(space)[4]
           else let val a = get(c,i,j)
                    fun f(x,y) = free(c,x,y) orelse get(c,x,y) <> a
                    val g = case (f(i-1,j),f(i+1,j),f(i,j-1),f(i,j+1))
                            of (false,false,false,false)
                                   => interior(pos(c,a))
                            |    (true,_,_,true) => corner
                            |    (true,_,true,_) => corner
                            |    (false,_,true,_) => horiz_edge
```

```
                              |     (false,_,_,true) => horiz_edge
                              |     (true,false,_,_) => left_edge
                              |     (false,true,_,_) => right_edge
                              |     (true,true,_,_) => column
           in g(a)(i,j) end

end (* Display *)
```

In the next section, *Display* is used for an ASCII representation of container fillings. An alternative functor *DisplayPS*, which produces PostScript code, will be defined in Section 8.

[4]For functions qualified by the structure name Aux, cf. Section 7.

5 Instances of *Container* and *Display*

After having defined the loading strategy and the display procedure for containers in a generic way by the functors *Container* and *Display*, we provide three classes, *CONTAINER, SHELF* and *SHELFSYSTEM*, each including an instance of *Container*, called *Build*, and an instance of *Display*, called *Draw*. The three versions of *Build* and *Draw* differ in the respective actualizations of the formal parameter *Obj* of *Container* and the formal parameter *Picts* of *Display*.

Furthermore, the classes *CONTAINER, SHELF* and *SHELFSYSTEM* provide a pointer *PERMS* to a stream of permutations of an object list. A call of the function *init* creates a stream of all orders of the set of objects to be loaded. A call of the function entails three actions:

- the removal of a given number of permutations from the stream,
- a call of *Build.fill* with the first permutation of the remaining stream as the object order the loading strategy starts with,
- a call of *Draw.container* with the load returned by *Build.fill*.

The first class *CONTAINER* does not impose any further constraints on the loading except those already defined in *Container*:

```
functor CONTAINER() = struct

structure Object = struct type obj = int * int * int
                          fun le(_,l,_) = l
                          fun he(_,_,h) = h
                          fun topConstraint _ = true
                          val Aligned = false
                          val Symmetric = false
                   end

structure Build = Container(structure Obj = Object)

structure Draw = Display(structure Picts =
   struct open Build
          val space = 3
          fun interior _ _ _ = "###"
          fun corner(a:int,_,_) _ = Aux.left(makestring(a),"-",3)
          fun horiz_edge _ _ = "---"
          fun left_edge _ _ = "|##"
          fun right_edge _ _ = "##|"
          fun column _ _ = "|#|"
   end)

local val PERMS = ref(Aux.mt:(int*int*int) list Aux.stream)
      val SIZE = ref(0,0)

in fun  init(objects,l,h) = (SIZE:=(l,h); PERMS:=Aux.permutations(objects))
```

```
   fun  fill(n) = let val objects = (PERMS:=Aux.suffix(!PERMS,n);
                                     Aux.head(!PERMS))
                      val [c] = Build.fill(objects)[!SIZE]
                  in Draw.container(c,"CONTAINER") end
end

end (* CONTAINER *)
```

Shelves are loaded with books or HiFi components. Hence the formal parameter *Obj* of *Container* is actualized by a suitable structure *BookOrHifi*. The top constraint of *BookOrHifi* demands that all objects put on top of other objects are books lying on their cover, i.e. their height must not exceed their length.

```
datatype object_type = book | hifi

functor SHELF() = struct

structure BookOrHifi = struct type obj = int * int * int * object_type
                              fun le(_,l,_,_) = l
                              fun he(_,_,h,_) = h
                              fun topConstraint(_,l:int,h,t)
                                    = l >= h andalso t = book
                              val Aligned = false
                              val Symmetric = true
                       end

structure Build = Container(structure Obj = BookOrHifi)

structure Draw = Display(structure Picts =
   struct open Build
          val space = 3
          fun interior _ (_,_,_,book) _ = "###"
          |   interior _ _ _ = "@@@"
          fun corner(a:int,_,_,_) _ = Aux.left(makestring(a),"-",3)
          fun left_edge(_,_,_,book) _ = "|##"
          |   left_edge _ _ = "|@@"
          fun right_edge(_,_,_,book) _ = "##|"
          |   right_edge _ _ = "@@|"
          fun column(_,_,_,book) _ = "|#|"
          |   column _ _ = "|@|"
   end)

local val PERMS = ref(Aux.mt:(int*int*int*object_type) list Aux.stream)
      val SIZE = ref(0,0)

in fun  init(objects,l,h) = (SIZE:=(l,h); PERMS:=Aux.permutations(objects))

   fun  fill(n) = let val objects = (PERMS:=Aux.suffix(!PERMS,n);
                                     Aux.head(!PERMS))
                      val [shelf] = Build.fill(objects)[!SIZE]
```

```
                  in Draw.container(shelf,"SHELF") end
end

end (* SHELF *)
```

Shelf systems are loaded with shelves. Hence the formal parameter *Obj* of *Container* is actualized by an object *SH* of the class *SHELF*. The type *obj* passed to *Container* is given by *SH.Build.cont ref.* Hence, as an object of a shelf system, a shelf is identified by a pointer to it.

```
functor SHELFSYSTEM() = struct

structure SH = SHELF()

structure Shelf = struct type obj = SH.Build.cont ref
                         val le = SH.Build.len o !
                         val he = SH.Build.hei o !
                         fun topConstraint _ = true
                         val Aligned = true
                         val Symmetric = false
                  end

structure Build = Container(structure Obj = Shelf)
```

For displaying a shelf system the position of each of its shelves must be accessible by the *SHELFSYSTEM* instance of *Display.* In each call of the function *interior(x,y)(a)* within a call *container(c), (x,y)* is the position of *a* in *c* (cf. Section 4). If *a* is a shelf and *c* is a shelf system, then the correct position of an object *b* of *a* within *c* is obtained by adding *(x-1,y-1)* to the relative position *(i,j)* of *b* in *a.* Conversely, position *(i,j)* in *c* occupies that item, which occupies position *(i-x+1,j-y+1)* in *a.* Due to the module structure of our program we can perform this geometric translation directly by calling the *SHELF's* function *picture*, translating the result and assigning it to *SHELFSYSTEM's* function *interior*:

```
structure Draw = Display(structure Picts =
   struct open Build
          val space = 3
          fun interior(x,y)(sh)(i,j) = SH.Draw.picture(!sh)(i-x+1,j-y+1)
          fun corner _ _ = "+++"
          fun horiz_edge _ _ = "+++"
          fun left_edge _ _ = "+  "
          fun right_edge _ _ = "  +"
          fun column _ _ = "+ +"
   end)
```

Furthermore, *SHELFSYSTEM* contains a pointer *objPERMS* to a stream of object orders and a pointer *shelfPERMS* to a stream of shelf orders. *init* initializes *!SIZE* and the streams. *fill(n,k)* removes n permutations from *!objPERMS* and k permutations from *!shelfPERMS* and then fills the head of the rest of *!objPERMS* into the head of the rest of *!shelfPERMS.* The filled shelves are framed (cf. Section 2) and loaded into a container with size *!SIZE.* This yields a shelf system.

```
local val objPERMS = ref(Aux.mt:(int*int*int*object_type) list Aux.stream)
```

```
      val shelfPERMS = ref(Aux.mt:(int*int) list Aux.stream)
      val SIZE = ref(0,0)

in fun  init(objects,shelves,l,h)
          = (SIZE:=(l,h); objPERMS:=Aux.permutations(objects);
                          shelfPERMS:=Aux.permutations(shelves))

   fun  fill(n,k)
          = let val objects = (objPERMS:=Aux.suffix(!objPERMS,n);
                               Aux.head(!objPERMS))
                val shelves = (shelfPERMS:=Aux.suffix(!shelfPERMS,k);
                               Aux.head(!shelfPERMS))
                val shelves = SH.Build.fill(objects)(shelves)
                val shelves = map(ref o SH.Build.frame)(shelves)
                val [shs] = Build.fill(shelves)[!SIZE]
            in Draw.container(shs,"SHELFSYSTEM") end
end

end (* **SHELFSYSTEM** *)
```

6 A Design Session

We define three structures by calling the functors *CONTAINER, SHELF* and *SHELFSYSTEM*. Each of these structures provides a pointer *PERMS* (respectively two pointers *objPERMS* and *sizePERMS* in the case of *SHELFSYSTEM*) to a stream of permutations, which are initialized by a call of *init* and modified by subsequent calls of *next* (cf. Section 5).

```
structure C = CONTAINER()
structure SH = SHELF()
structure SHS = SHELFSYSTEM()
```

All objects we are going to load are given as values of the following function:

```
fun bh(1) = (1,3,6,hifi)
|   bh(2) = (2,5,1,book)
|   bh(3) = (3,2,2,book)
|   bh(4) = (4,5,3,book)
|   bh(5) = (5,4,2,book)
|   bh(6) = (6,1,1,book)
|   bh(7) = (7,1,4,book)
|   bh(8) = (8,2,2,book)
|   bh(9) = (9,3,8,hifi)
|   bh(10) = (10,3,3,book)
|   bh(11) = (11,6,3,book)
|   bh(12) = (12,3,5,hifi)
|   bh(13) = (13,3,3,book)
|   bh(14) = (14,2,7,book)
|   bh(15) = (15,2,6,book)
|   bh(16) = (16,5,3,book)

fun obj(n) = let val (a,l,h,t) = bh(n) in (a,l,h) end

val tenObjects = map(obj)(Aux.up(1,10))
val tenBooksAndHifis = map(bh)(Aux.up(1,10))
```

The command sequence *C.init(tenObjects,17,9); C.fill(0);* yields the following contents of *CONTAINER*:

```
9 | 1--------
8 | |#######|9--------
7 | |#######||#######|
6 | |#######||#######|            4--------------
5 | |#######||#######|            |#############|
4 | 1--------|#######|      7--   4--------------
3 | 10-------|#######|      |#|   2--------------
2 | |#######||#######|8-----|#|   5-----------3-----
1 | 10-------9--------8-----7--6--5-----------3-----
   ----------------------------------------------------
     1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
```

SH.init(tenBooksAndHifis,17,9); SH.fill(0); yields the following contents of *SHELF*:

```
9 |
8 |          9--------
7 |          |@@@@@@@|
6 |          |@@@@@@@|         5-----------   1--------
5 | 3-----   |@@@@@@@|         5-----------   |@@@@@@@|
4 | 3-----   |@@@@@@@|      7--4--------------|@@@@@@@|
3 | 10-------|@@@@@@@|6--   |#||#############||@@@@@@@|
2 | |#######||@@@@@@@|8-----|#|4--------------|@@@@@@@|
1 | 10-------9--------8-----7--2--------------1--------
   ----------------------------------------------------
     1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
```

With the command *SH.fill(100);* we start the filling process with the 100th element of the stream of all permutations of *tenBooksAndHifis*. We come up with the same loading except that the piling of objects 2 and 4 is exchanged. Next we pass the next 500 orders of *tenBooksAndHifis* and start the filling process with the permutation reached then. *SH.fill(500);* yields:

```
9 |
8 |                              9--------
7 |                              |@@@@@@@|
6 |               5-----------   |@@@@@@@|1--------
5 |          6--   5-----------   |@@@@@@@||@@@@@@@|
4 |          8-----4--------------|@@@@@@@||@@@@@@@|7--
3 | 10-------8-----|#############||@@@@@@@||@@@@@@@||#|
2 | |#######|3-----4--------------|@@@@@@@||@@@@@@@||#|
1 | 10-------3-----2--------------9--------1--------7--
   ----------------------------------------------------
     1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
```

Let's build shelf systems. First we put objects into two shelves.

```
val  twoShelves = [(15,7),(15,4)]
val  booksAndHifis1 = map(bh)[2,3,4,5,6,11,12,13,14,15,16]
```

A call of *SHS.init* initializes the stream of permutations of shelves as well as the stream of permutations of books and HiFi components. The command sequence *SHS.init(booksAndHifis1,twoShelves,17,15); SHS.fill(0,0);* yields the following contents of *SHELFSYSTEM*:

```
15 | +++++++++++++++++++++++++++++++++++++++++++++++++++
14 | +  2--------------                                +
13 | +  11----------------4--------------              +
12 | +  |################||#############|              +
11 | +  11----------------4--------------              +
10 | +++++++++++++++++++++++++++++++++++++++++++++++++++
 9 | +++++++++++++++++++++++++++++++++++++++++++++++++++
 8 | +                 6--   14----                    +
 7 | +                 15----|####|                    +
 6 | +  5-----------   |####||####|3-----   12-------  +
 5 | +  5-----------   |####||####|3-----   |@@@@@@@|  +
 4 | +  16-------------|####||####|13-------|@@@@@@@|  +
 3 | +  |#############||####||####||#######||@@@@@@@|  +
 2 | +  16-------------15----14----13-------12-------  +
 1 | +++++++++++++++++++++++++++++++++++++++++++++++++++
    ----------------------------------------------------
      1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
```

Next we start with the same object order, but with the two shelves exchanged, i.e. we call *SHS.fill(0,1);* and obtain:

```
15 | +++++++++++++++++++++++++++++++++++++++++++++++++++
14 | +        14----                                   +
13 | +  15----|####|                                   +
12 | +  |####||####|12-------5-----------              +
11 | +  |####||####||@@@@@@@|5-----------              +
10 | +  |####||####||@@@@@@@|4--------------           +
 9 | +  |####||####||@@@@@@@||#############|3-----     +
 8 | +  15----14----12-------4--------------3-----     +
 7 | +++++++++++++++++++++++++++++++++++++++++++++++++++
 6 | +++++++++++++++++++++++++++++++++++++++++++++++++++
 5 | +  2--------------                                +
 4 | +  16-------------13-------11----------------     +
 3 | +  |#############||#######||################|     +
 2 | +  16-------------13-------11----------------6--  +
 1 | +++++++++++++++++++++++++++++++++++++++++++++++++++
    ----------------------------------------------------
      1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17
```

Next we have four shelves and a different set of books and HiFi components:

```
val  fourShelves = [(5,8),(4,14),(5,9),(4,14)]
val  booksAndHifis2 = map(bh)[1,2,3,4,5,6,7,8,9,10,12,14]
```

SHS.init(booksAndHifis2,fourShelves,19,21); SHS.fill(0,0); yields:

```
21 | +++++++++++++++++++++
20 | +                   +
19 | +                   +
18 | +                   +
17 | +  5-----------     +                  ++++++++++++++++++
16 | +  5-----------     ++++++++++++++++++++                +
15 | +  2--------------  ++                ++                +
14 | +  4--------------  ++                ++                +
13 | +  |#############|  ++  3-----        ++                +
12 | +  4--------------  ++  3-----        ++                +
11 | ++++++++++++++++++++++  8-----        ++                +
10 | ++++++++++++++++++++++  8-----        ++                +
 9 | +  6--   10-------  ++  9--------     ++                +
 8 | +  14----|#######|  ++  |@@@@@@@|     ++                +
 7 | +  |####|10-------  ++  |@@@@@@@|     ++  1--------     +
 6 | +  |####|12-------  ++  |@@@@@@@|     ++  |@@@@@@@|     +
 5 | +  |####||@@@@@@@|  ++  |@@@@@@@|7--  ++  |@@@@@@@|     +
 4 | +  |####||@@@@@@@|  ++  |@@@@@@@||#|  ++  |@@@@@@@|     +
 3 | +  |####||@@@@@@@|  ++  |@@@@@@@||#|  ++  |@@@@@@@|     +
 2 | +  14----12-------  ++  9--------7--  ++  1--------     +
 1 | +++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    ----------------------------------------------------------
      1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
```

SHS.fill(1,11); yields:

```
21 | +++++++++++++++++++++
20 | +                   +
19 | +                   +
18 | +  5-----------     +
17 | +  5-----------     +++++++++++++++++++
16 | +  4--------------  ++                +++++++++++++++++++
15 | +  |#############|  ++                ++                +
14 | +  4--------------  ++                ++                +
13 | +  2--------------  ++                ++                +
12 | ++++++++++++++++++++++                ++                +
11 | ++++++++++++++++++++++  3-----        ++                +
10 | +  8-----           ++  3-----        ++                +
 9 | +  8-----10-------  ++  9--------     ++                +
 8 | +  14----|#######|  ++  |@@@@@@@|     ++                +
 7 | +  |####|10-------  ++  |@@@@@@@|     ++  1--------     +
 6 | +  |####|12-------  ++  |@@@@@@@|6--  ++  |@@@@@@@|     +
 5 | +  |####||@@@@@@@|  ++  |@@@@@@@|7--  ++  |@@@@@@@|     +
 4 | +  |####||@@@@@@@|  ++  |@@@@@@@||#|  ++  |@@@@@@@|     +
 3 | +  |####||@@@@@@@|  ++  |@@@@@@@||#|  ++  |@@@@@@@|     +
 2 | +  14----12-------  ++  9--------7--  ++  1--------     +
 1 | +++++++++++++++++++++++++++++++++++++++++++++++++++++++++
    ----------------------------------------------------------
      1  2  3  4  5  6  7  8  9  10 11 12 13 14 15 16 17 18 19
```

7 Streams, Traversals and String Primitives

The following module *Aux* provides auxiliary types and functions used by the functors *Container, Display, DisplayPS* (cf. Section 8 below) or their instances.

```
structure Aux = struct

fun     up(k,0) = nil
|       up(k,n) = k::up(k+1,n-1)

fun     sum_of_prods(w) = fold(op +)(map(op * )(w))(0)

fun     iterate(str)(0) = ""
|       iterate(str)(n) = str^iterate(str)(n-1)
```

The crucial data structure for generating object orders is the polymorphic data type *'a stream*, which allows us to stepwise enumerate the elements of a long list. This type is easily implemented in ML by employing functions of type *unit* $\to$ *'a stream.* Such a function *s* represents an "unevaluated" stream whose evaluation is postponed until an occurrence of the call *s().*

```
infix  & %

datatype 'a stream = mt | & of 'a * (unit -> 'a stream)

fun     nil%s = s()
|       (x::w)%s = x&(fn()=>w%s)

exception emptyStream

fun     head(x&s) = x
|       head(mt) = raise emptyStream

fun     tail(x&s) = s()
|       tail(mt) = raise emptyStream

fun     suffix(s,0) = s
|       suffix(x&s,n) = suffix(s(),n-1)
|       suffix _ = raise emptyStream

fun     mapconc(f)(mt) = mt
|       mapconc(f)(x&s) = let fun s1() = mapconc(f)(s()) in f(x)%s1 end
fun     permutations(nil) = nil&(fn()=>mt)
|       permutations(x::w) = mapconc(insert(x))(permutations(w))

and     insert(x)(w) = let fun f(v,x,nil) = [v@[x]]
                           |   f(v,x,y::w) = (v@x::y::w)::f(v@[y],x,w)
                         in f(nil,x,w) end
```

Functions with the one-element range type *unit* are usually called procedures. In particular, we used procedures for traversing argument intervals of a function *f* up to a point where *f* is undefined (cf. Section 3).

```
fun     forward(f)(i)(x) = if i > x then () else (f(i); forward(f)(i+1)(x))


fun     down(f)(i,j)(x,y)
          = if y < j then () else (forward(fn(i)=>f(i,y))(i)(x);
                                   down(f)(i,j)(x,y-1))
```

forward checks the one-dimensional interval *[i,x]* from the least up to the greatest element. *down* traverses the two-dimensional interval *[(i,j),(x,y)]* from the greatest up to the least pair.

```
fun     left(str,ch,space) = str^iterate(ch)(space-String.length(str))


fun     right(str,ch,space) = iterate(ch)(space-String.length(str))^str


fun     center(str,ch,space) = let val rest = space-String.length(str)
                                   val back = rest div 2
                                   val even = rest mod 2 = 0
                                   val front = back+(if even then 0 else 1)
                               in iterate(ch)(front)^str^iterate(ch)(back)           end


fun     blanks(n) = iterate" "(n)
fun     hyphens(n) = iterate"-"(n)


fun     Ints2Strings(space,n)
          = let fun f(i,s) = center(makestring(i:int)," ",space)^s
            in fold(f)(up(1,n))"" end


fun     newline(space,i) = "\n"^right(makestring(i:int)," ",space)^" | "


fun     columns(space,n)
          = "\n"^blanks(space+2)^"-"^iterate(hyphens(space))(n)^
            "\n"^blanks(space+3)^Ints2Strings(space,n)


and     ruler(f)(space,file,l,h)
         = let fun g(1,j) = output(file,newline(space,j)^f(1,j))
                 |   g(i,j) = output(file,f(i,j))
           in down(g)(1,1)(l,h); output(file,columns(space,l)^"\n\n") end
```

newline places row indices at the beginning of a row. *columns* prints a final row of column indices. *ruler* prints the values of a binary function *f* and embeds them into a coordinate system.

```
fun     factors(x,y,l,h,space,scale)
          = makestring(real(x*space-space)+scale)^" "^
            makestring(real(y*space-space)+scale)^" "^
            makestring(real(l*space)-scale-scale)^" "^
            makestring(real(h*space)-scale-scale)
```

```
fun    placeID(x,y,a:int,l,space)
         = "0 setgray "^
           makestring(x*space-12)^" "^makestring(y*space-12)^" moveto\n"^
           "("^center(makestring(a)," ",3*l)^") show\n"

end    (* Aux *)
```

8 PostScript Output

How do we have to change *Display* (cf. Section 4) and the classes *CONTAINER, SHELF* and *SHELFSYSTEM* such that the loadings are translated into PostScript code instead of ASCII files? Again, the signature *Pictures* imports container functions, which are needed for building the output. But now there are only three code parameters: *linewidth, linegray* and *interior.* We need no longer dip to the level of point descriptions as in the case of ASCII pictures.

```
signature Pictures = sig  type obj and cont
                          val le : obj -> int
                          val he : obj -> int
                          val len : cont -> int
                          val hei : cont -> int
                          val Objs : cont -> obj list
                          val pos : cont * obj -> int * int
                          val space : int
                          val linewidth : real
                          val linegray : real
                          val interior : int * int -> obj -> string
                     end
```

linewidth and *linegray* yield the width resp. gray colour of object edges. As in *Display, interior* determines the way how and where an object is drawn. The string value of this function comprises appropriate PostScript code.

```
functor DisplayPS(structure Picts : Pictures) = struct open Picts

fun    container(c,file)
         = let val file = open_out(file^".eps")
               val code = "%!PS-Adobe-3.0 EPSF-3.0\n"^
                          "%%BoundingBox: 5 5 "^
                          makestring(len(c)*factor+20)^" "^
                          makestring(hei(c)*factor+20)^"\n"^
                          "/Helvetica 10 selectfont\n"^
                          "1 setlinejoin\n"^
                          makestring(space)^" "^
                          makestring(space)^" translate\n"^
                          contCode(c)^objsCode(c)(Objs(c))^"showpage"
           in output(file,code); close_out(file) end

and    contCode(c) = "0.2 setgray 0 0 "^
                     makestring(len(c)*space)^" "^
                     makestring(hei(c)*space)^" rectstroke\n"

and    objsCode(c,nil) = ""
|      objsCode(c,a::objs)
         = let val scale = linewidth/2.0
               val (x,y) = pos(c,a)
```

```
              val (l,h) = (le(a),he(a))
          in interior(x,y)(a)^
             makestring(linewidth)^" setlinewidth\n"^
             makestring(linegray)^" setgray\n"^
             Aux.factors(x,y,l,h,space,scale)^" rectstroke\n"^
             objsCode(c)(objs) end

fun    translate(x,y)(c)
         = makestring(x*space)^" "^makestring(y*space)^" translate\n"^
           objsCode(c)(Objs(c))^
           "-"^makestring(x*space)^" -"^makestring(y*space)^" translate\n"

end    (* DisplayPS *)
```

Furthermore, only the structure *Draw*, which occurs in each of the classes *CONTAINER, SHELF* and *SHELFSYSTEM* (cf. Section 5), needs to be changed, namely into a call of the above functor *DisplayPS*. In *CONTAINER, Draw* is now defined as follows:

```
structure Draw = DisplayPS(structure Picts =
   struct open Build
          val space = 15
          val linewidth = 1.0
          val linegray = 0.0
          fun interior(x,y)(a,l,h,t)
                = "0.7 setgray "^ Aux.factors(x,y,l,h,15,1.0)^
                    " rectfill\n"^Aux.placeID(x,y+(h div 2),a,l,15)
   end)
```

SHELF gets almost the same definition of *Draw*:

```
structure Draw = DisplayPS(structure Picts =
   struct open Build
          val space = 15
          val linewidth = 1.0
          val linegray = 0.0
          fun interior(x,y)(a,l,h,t)
                = (if t = hifi then "0.3 setgray " else "0.7 setgray ")^
                    Aux.factors(x,y,l,h,15,1.0)^" rectfill\n"^
                    Aux.placeID(x,y+(h div 2),a,l,15)
   end)
```

The main task of *Draw* in *SHELFSYSTEM* is to geometrically place the contents of a shelf into a shelf system. In Section 5 we described the necessary translation. PostScript code handles lines and planes and not only points. We need no longer define the translation as a pointwise transformation. However the hierarchic access from *SHELFSYSTEM* to *SHELF* via the structure *SH* (cf. Section 5) is still the crucial point, which allows us to define the interior of an object of a shelf system in terms of *SH.Draw*:

```
structure Draw = DisplayPS(structure Picts =
   struct open Build
```

```
        val space = 15
        val linewidth = 15.0
        val linegray = 0.6
        fun interior(x,y) = SH.Draw.translate(x-1,y-1) o !
   end)
```

We repeat the design session of Section 6, now along with the contents of the respective *eps* file after this has been evaluated by a PostScript interpreter.

C.init(tenObjects,17,9); C.fill(0);

SH.init(tenBooksAndHifis,17,9); SH.fill(0);

SH.fill(600);

SHS.init(booksAndHifis1,twoShelves,17,15); SHS.fill(0,0);

SHS.fill(0,1);

SHS.init(booksAndHifis2,fourShelves,19,21); SHS.fill(0,0);

SHS.fill(1,11);

9 Conclusion

We have given an example of how the concepts of functional programming such as polymorphism, functional abstraction, module abstraction (using functors), partiality (using exceptions) and stream handling can be employed for solving a hierarchical configuration problem. The task of building shelf systems falls into several concerns, which have been tackled separately. The corresponding modules were defined as functors. Each module consists of simple code, which is due to the fact that the underlying algorithms could be implemented directly on abstract data types. The places where a module is used are easily identified because they are encapsulated into structures defined as functor calls. With parameters of a functor we realize module hierarchies, even recursive ones if the same module appears twice in the hierarchy. For instance, *SHELFSYSTEM* includes a call of the functor *Container* with an actual parameter, which stems from a further call of *Container* (cf. Section 5). This recursion did not not come up as a subtle implementation detail, it is already inherent to the problem we were going to implement.

Functional programming aims at avoiding dynamic objects. Dynamic objects make a program difficult to verify because they change their values at runtime and thus the correctness analysis has to consider all runtime states the program can achieve. The object-oriented programming paradigm takes the converse position and makes every object a dynamic one by claiming that this is the natural view. We think the truth is in the middle. A program using static objects is indeed much easier to understand, even if one dispenses with a formal proof. Moreover, there are a number of problems, which have direct functional, static solutions. Most of the computation-intensive stuff, which resides at the bottom levels of a program, is of this sort. However, there may be high levels where states do not only come up with the implementation, but are part of the problem. Especially when interaction is involved, dynamic objects are not only preferable, but even necessary. This is exactly the point where we have introduced dynamic objects into our program. As nullary functors, *CONTAINER, SHELF* and *SHELFSYSTEM* are in fact classes, while a call of one of these functors causes the creation of two objects: the permutation stream *PERMS* and a number pair *SIZE*. The user changes their states by calls of the functions *init* and *fill* (cf. Sections 5 and 6). Object-oriented programming complies well with functional programming.

A high degree of modularization, the use of abstract data types and the focus on static objects enhances the flexibility and reusability of a program. A module may be redefined, an algorithm may be changed, and this will hardly affect other parts of the program. *CONTAINER, SHELF* and *SHELFSYSTEM* are variants of each other, with increasing complexity. In object-oriented terms, the three classes are heirs of the functors *Container* and *Display*. We wanted to change the output from ASCII to PostScript code and achieved this goal just by replacing *Display* with *DisplayPS* and corresponding replacements of the calls of *Display* in *CONTAINER, SHELF* and *SHELFSYSTEM* (cf. Section 8). All this is completely independent of the main algorithm involved: the loading strategy (cf. Section 3). In fact, a number of variants of this algorithm might be worthwhile to be analyzed. Our program would provide a flexible environment for such experiments. For instance, if the number of objects, which are unpacked by one call of *unFill*, is increased, the enumeration of object orders is changed and thus the loading algorithm may come up with other fillings and a different time behaviour. Or what about genetic variants of the algorithm?

10 Acknowledgement

I am grateful to Ana Pasztor and Mihaly Lenart for directing my attention to the field of configuration problems that are an important part of engineering design. Our joint work started with [LPP 93, 93a] and pursues the overall goal of applying and evaluating logic-based concepts and methods within this area. Thanks to Mihaly for commenting on a previous version of this paper.

References

[BW 88] R. Bird, Ph. Wadler, *Introduction to Functional Programming*, Prentice-Hall 1988

[Dav 92] A.J.T. Davie, *An Introduction to Functional Programming Systems using Haskell*, Cambridge University Press 1992

[DJ 90] N. Dershowitz, J.-P. Jouannaud, *Rewrite Systems*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Elsevier (1990) 243-320

[MQu 84] D. MacQueen, *Modules for Standard ML*, Proc. ACM Conf. on LISP and Functional Programming, Austin (1984) 198-207

[GH 93] J.V. Guttag, J.J. Horning, *Larch: Languages and Tools for Formal Specification*, Springer 1993

[LPP 93] M. Lenart, P. Padawitz, A. Pasztor, *Automating Creative Design*, Proc. AAAI 1993 Spring Symposium on AI and Creativity, Stanford University (1993) 67-71

[LPP 93a] M. Lenart, P. Padawitz, A. Pasztor, *Formal Specifications for Design Automation*, Proc. IFIP WG 5.2 Workshop in Formal Design Methods for Computer-Aided Design, Tallinn, Estonia (1993) 169-187

[Pad 92] P. Padawitz, *Deduction and Declarative Programming*, Cambridge University Press 1992

[Pad 93] P. Padawitz, *Programmierung*, Course notes (in German), FB Informatik, Universität Dortmund 1993

[Pad 94] P. Padawitz, *Expander: A System for Testing and Verifying Functional-Logic Programs*, Report No. 522/1994, FB Informatik, Universität Dortmund

[Pau 91] L.C. Paulson, *ML for the Working Programmer*, Cambridge Unversity Press 1991

[Rea 89] C. Reade, *Elements of Functional Programming*, Addison-Wesley 1989

[Tur 90] D.A. Turner, *An Overview of Miranda*, in: D.A. Turner, ed., Research Topics in Functional Programming, Addison-Wesley 1990

[Ull 94] J.D. Ullman, *Elements of ML Programming*, Prentice-Hall 1994

[Wir 90] M. Wirsing, *Algebraic Specification*, in: J. van Leeuwen, ed., Handbook of Theoretical Computer Science, Vol. B, Elsevier (1990) 675-788