Computing Rectangular Dissections

A Case Study in Deriving Functional Programs
from Logical Specifications

March 22, 1996

Peter Padawitz
Fachbereich Informatik
D-44221 Universität Dortmund, Germany
peter@ls5.informatik.uni-dortmund.de

Abstract

Logical specifications provide an abstract level of programming where programs are given as axioms defining functions and predicates on constructor-based data types. Axiomatic function definitions are transformed into functional programs, predicate definitions are compiled into logic programs. If functions are to be implemented as logic programs, they must be *flattened* into predicates. In this paper, we exemplify the inverse translation: predicates used as multi-valued, nondeterministic, functions are compiled into stream-generating functions, which enumerate multiple values. Generate-and-test algorithms, usually implemented as logic programs, can be realized in this way as functional programs. Our case study deals with a configuration problem. Two nondeterministic algorithms that compute rectangular dissections are specified in the functional-logic language of *Expander*, flattened into *Prolog* programs, and, alternatively, translated into *ML* programs, which produce streams of dissections. Besides the compilation of predicates into stream functions the case study illustrates the use of polymorphic types and higher-order functions for accomplishing well-structured and reusable software.

Contents

1 Introduction **2**

1.1 Multi-valued functions 2

1.2 The stream data type 4

1.3 Overview 6

2 Building and using a specification **6**

2.1 A dissection algorithm 6

2.2 Expander versus Prolog 10

2.3 Tests and proofs 13

2.4 Dimensioning dissections 16

3 Deriving functional code **18**

3.1 Computing dissections 18

3.2 Drawing dissections 21

3.3 Sample dissection streams 22

4 A shape grammar solution **22**

4.1 The specification 23

4.2 The functional program 26

5 Conclusion **28**

1 Introduction

1.1 Multi-valued functions

In [Car 93], Carlson presents a program for dissecting rectangles. It is written in a Prolog dialect, called Grammatica. Besides Prolog constructs Grammatica includes a constraint solver and an evaluator for regular expressions used to represent nondeterministic functions. We think that - apart from special-purpose constraint solvers - the logic and functional languages, which are currently in use or under development, are suitable for both the abstract specification and the efficient implementation of nondeterministic algorithms. A logic program defining relations without functional dependencies mostly realizes a nondeterministic algorithm or a multi-valued function. More precisely, suppose a program P defines a relation R with arguments taken from n data domains $D_1, \ldots, D_n$. R is a subset of the Cartesian product $D = D_1 \times \ldots \times D_n$ and we say that for an n-tuple $d = (d_1, \ldots, d_n) \in D$, the atom Rd holds true if $d \in R$, while Rd does not hold if $d \notin R$. P is just a finite presentation of the possibly infinite set R. The elements of $D_1, \ldots, D_n$ comprise the actual parameters of P.

In functional and imperative programs we distinguish from the beginning between input parameters on the one hand and output or result parameters on the other hand. This differs from a logic program P. Which sets among $D_1, \ldots, D_n$ are input and which ones are output of P depends on the call of P or the *query* to R. If, for

instance, R is part of a database, D_1 is a set of object identifiers and $D_2, \ldots, D_n$ represent object attributes, then $R(d_1, \ldots, d_n)$ holds true if $d_2, \ldots, d_n$ denote attribute values of d_1. Given a variable x, a typical query to R has the form

$$R(d_1, ..., d_{i-1}, x, d_{i+1}, ..., d_n). \quad (1)$$

As a call of P (1) uses $D_1, \ldots, D_{i-1}, D_{i+1}, \ldots, D_n$ as input domains and D_i as an output domain and asks for $d_i \in D_i$ such that $R(d_1, \ldots, d_n)$ holds true. Whatever the query means, whatever the actual input and output parameters of P are, the compiler or interpreter of a logic programming language is a procedure that *takes* input and *produces* output. In contrast to a functional program a logic program P may realize several *input-output modes* of the same relation. But each call of P selects one of them.

This view of logic programming motivates the main issue of this paper: a translation of logic into functional programs. We restrict ourselves to logic programs with fixed input-output modes of the relations they realize. If the program P implements the relation R, we only admit those queries to R, which use the same domains as input and output domains, respectively. All admissable queries are of the form (1) with fixed i. The task of P is to solve (1) in x for all $d_j \in D_j, 1 \leq j \leq n, j \neq i$. Hence the corresponding functional program will include a procedure for enumerating all solutions of (1). Since there may be infinitely many, we use *streams* to store them. Streams are also called *lazy lists* because they can be used in a program even at places where - at runtime - most of their elements have not yet been computed. Streams can be used both for *don't-care nondeterminism* (get *one* solution) and for *don't-know nondeterminism* (get *all*solutions).

We specify multi-valued functions first in the functional-logic language of our prototyping system *Expander* (cf. [Pad 94]). The dissection example is used for presenting three ways to evaluate a set F of multi-valued functions. First, the Expander specification can be excuted directly. Secondly, F can be flattened into a set of relations (cf. [BGM 88]) and we come up with a pure logic program written in *Prolog* (cf. Section 2.2). Thirdly, we derive a functional program from the specification. As a suitable language for functional programs we use *Standard ML* (cf. e.g. [Pau 91]). A little familiarity with Prolog and ML is assumed. Let us start with two simple examples.

Example 1.1.1 *Get an element from a list* In Expander a logic program for selecting an element of a list is given by the axioms of the following Horn clause specification.

```
GET
      preds     get 1 2
      vars      x y L
      axioms    (1) { get(x::L,x) }
                (2) { get(x::L,y) } <== { get(L,y) }
      conjects  (1) { get([2,4,3,7,9],x) }
```

`preds (functs)` precedes the list of relations (functions) to be defined. Each relation or function symbol is followed by a list of numbers of those axioms, which define the function or relation. Each axiom is a Horn clause with the implication arrow `<==` separating its conclusion from its premise. :: is the binary operation, which appends its first argument to the list given by its second argument.[1] The `conjects` part contains sample queries.[2] For testing the program for `get` we ask Expander to solve Conjecture 1 whereafter we obtain the following output:

```
      (1) { x = 2 }
      (2) { x = 4 }
```

[1] The notation of lists and list funcitons is taken over from ML.

[2] In general, arbitrary Gentzen clauses are allowed here each of which is to be proved or tested by solving its conclusion under the constraint given by its premise.

```
(3) { x = 3 }
(4) { x = 7 }
(5) { x = 9 }
```

The query `get([2,4,3,7,9],x)` fixes an input-output mode of `get`. Following the parallel logic language PARLOG (cf. [Gre 87]) we may denote the mode of `get` as $(L?, x!)$. ? stands for input, ! for output parameters. $(L?, x!)$ admits only queries of the form `get(L,x)` where L is a given list and x is a variable to be instantiated. □

Example 1.1.2 *Choose a natural number not less than n*

```
NAT
      preds     nat 1 2
      vars      x y
      axioms    (1) { nat(x,x) }
                (2) { nat(x,y) } <== { nat(s(x),y) }
      conjects  (1) { nat(5,x) }
```

`s` is the unary function mapping a natural number n to its successor $n+1$. Conjecture 1 fixes the mode of `nat` as $(x?, y!)$. If requested to solve the formula `nat(5,x)` Expander responds with the following output:

```
(1) { x = 5 }
(2) { x = 6 }
(3) { x = 7 }
(4) { x = 8 }
(5) { x = 9 }
(6) { x = 10 }
. . .
```

Of course, Expander's solving procedure is equipped with a maximal number of inference steps that are carried out for deriving solutions of `nat(5,x)`. The search for solutions finishes as soon as this number is exceeded. □

A functional program derived from a logical specification should compute solutions of each admissable query. Compilers for logic programs like the Warren machine (cf. [War 83]) achieve this goal by translating the programs into deterministic assembler code. However, such compilers transform the program independently of input-output modes and thus do not take into account our assumption that each defined relation has a fixed input-output mode. Under this assumption we are still faced with two questions:

- Which types are suitable for representing sets of solutions?
- In which order shall solutions be generated?

Ex. 1.1.1 suggests a list representation, but already 1.1.2 shows that list types are not appropriate in general because the complete solution set of `nat(5,x)` is infinite. Many applications exclude list representations even if the solution sets are finite, but very large. Hence ML and other functional languages provide *stream* types, which are more suitable for representing both infinite and large finite solution sets.

1.2 The stream data type

Streams are lists whose elements are evaluated only on demand. In ML, we define the type of streams with elements of type `'a` as follows:

```
infix &
datatype 'a stream = Nil | & of 'a * (unit -> 'a stream)
```

The type definition determines the objects of the type. A stream is either `Nil` (the empty stream) or a term of the form `x&s` where `x` is an element of type `'a` and `s` is a function of type `unit -> 'a stream`. Only the first element of a nonempty stream is given explicitly. The rest remains unevaluated in `s` and gets evaluated only if `s` is called. Since `unit` is ML's one-element type, there is only one call of `s`, namely `s()`, which returns either `Nil` or the second element of the original stream `x&s`. n elements of `x&s` are evaluated by the function `front`:

```
fun front(s,0) = nil
|   front(Nil,n) = nil
|   front(x&s,n) = x::front(s(),n-1)
```

Note the difference beween the empty stream `Nil` and the empty list, denoted by `nil`.

The delayed evaluation of streams is taken into account by stream operators such as the concatenation `%` of two streams, the filter of a substream and several map functions, which apply a function f to all elements of a list or stream and produce streams of f-images:

```
fun Nil % s = s
|   (x&s) % s' = x & (fn()=>s() % s')

fun MapL(f)(nil) = Nil
|   MapL(f)(x::L) = f(x) & (fn()=>MapL(f)(L))

fun MapS(f)(Nil) = Nil
|   MapS(f)(x&s) = f(x) % (fn()=>MapS(f)(s()))

fun MapLS(f)(nil) = Nil
|   MapLS(f)(x::L) = f(x) % (fn()=>MapLS(f)(L))

fun Filter(p)(Nil) = Nil
|   Filter(p)(x&s) = let fun s1() = Filter(p)(s())
                     in if p(x) then x&s1 else s1() end
```

Example 1.2.1 *Get an element from a list* Using the above stream type, `GET` (cf. Ex. 1.1.1) is transformed into the following ML function:

```
fun get(nil) = Nil
|   get(x::L) = x & (fn()=>get(L))
```

An expression of the form `fn()=>e` is an object of type `unit -> 'a stream`. `fn()=>e` denotes the function whose call `(fn()=>e)()` leads to the evaluation of `e`. Given a list `L`, the call `front(get(L),n)` computes the first n elements of the stream `get(L)`. For instance, the ML system evaluates `front(get[2,4,3,7,9],4)` to `[2,4,3,7]`. □

Example 1.2.2 *Choose a natural number not less than n* Using the stream type, `NAT` (cf. Ex. 1.1.2) is transformed into the following ML function:

```
fun nat(x) = x & (fn() => nat(x+1))
```

Given a natural number `x`, the call `front(nat(x),n)` computes the first n elements of the stream `nat(x)`. For instance, the ML system evaluates `front(nat(5),6)` to `[5,6,7,8,9,10]`. □

1.3 Overview

In the following chapter, we specify a new dissection algorithm in terms of an Expander specification DISSECT (Section 2.1). As mentioned above, Expander provides an integrated functional-logic language. For comparing such a language with logic languages we translate the specification DISSECT into Prolog and use common tricks for transforming functional into as logic programs (Section 2.2). Tests and proofs of the algorithm are carried out directly on the specification level (Section 2.3). The three ways of dimensioning dissections discussed in [Car 93] define several instances of the constraint C referred to in DISSECT and lead to the extension DISSECT_E of DISSECT (Section 2.4). In Section 3.1, we derive an ML program from DISSECT_E and apply the above-sketched schema of building streams of function values. Section 3.2 augments the ML program with a compiler of dissection streams into PostScript code so that one can visualize the computed streams. Section 3.3 provides test results.

In Chapter 4 we review the dissection algorithm given in [Car 93], which involves more nondeterminism than ours. Presented as a finite automaton over a suitable state space, it can be optimized to a great extent. Nevertheless, Carlson's algorithm remains ambiguous in the sense that many dissections are computed several times. Our algorithm works bottom-up by *merging* points and thus constructing the rectangles of a dissection. Carlson's algorithm works top-down insofar as a given plane is *decomposed* stepwise into increasingly smaller rectangles. Consequently, our algorithm is fast for dissections consisting of small rectangles, while Carlson's is more efficient if all dissections consist of a few big rectangles.

Both algorithms are developed from an Expander specification down to a functional ML program using the stream data type defined in Section 1.2.

2 Building and using a specification

2.1 A dissection algorithm

Our algorithm for computing rectangle dissections is derived from the functional program for building list partitions given in [BW 88], p. 135. As an Expander specification (cf. Section 1) this program reads as follows.

Example 2.1.1 *Compute all partitions of a list into sublists*

```
PARTITION
        preds    part 1 2 3
        vars     x y L L' P
        axioms   (1) { part([x],[[x]]) }
                 (2) { part(x::y::L,[x]::P) } <== { part(y::L,P) }
                 (3) { part(x::y::L,(x::L')::P) } <== { part(y::L,L'::P) }
        conjects (1) { part([1,2,3,4],P) }
```

The first argument of `part` is the list L that shall be splitted into sublists. The second argument yields a partition of L. We fix the mode of `part` as $(L?, P!)$. Axiom 1 defines `part` in the case where L consists of a single element. Then the only partition of L has L as its single element. Axioms 2 and 3 deal with lists starting with two elements. Here L stands for the rest of the list. Both axioms render `part` a nondeterministic function (cf. Section 1). They remove the first element x from the list x::y::L, compute a partition P (resp. L'::P) of the

remaining list y::L and extend P (L'::P) to a partition of x::y::L by appending the sublist [x] ([x::L']) to P. If asked to solve Conjecture 1, Expander produces the following output:

```
(1) { P = [[1,2,3,4]] }
(2) { P = [[1,2,3],[4]] }
(3) { P = [[1,2],[3,4]] }
(4) { P = [[1,2],[3],[4]] }
(5) { P = [[1],[2,3,4]] }
(6) { P = [[1],[2,3],[4]] }
(7) { P = [[1],[2],[3,4]] }
(8) { P = [[1],[2],[3],[4]] }
```

□

Dissections of a rectangle R are represented as partitions of the list of all points (x, y) covered by R. The following function `interval` returns all points of the two-dimensional interval between the least point $(x1, y1)$ and the greatest point $(x2, y2)$ of R.

```
interval(x1,y1)(x2,y2) = ite(gt(y1,y2),nil,row(x1,x2,y1)@interval(x1,s(y1))(x2,y2))
row(x1,x2,y) = ite(gt(x1,x2),nil,(x1,y)::row(s(x1),x2,y))
```

Expander partially evaluates standard functions and predicates such as `ite` (if-then-else), `gt` (greater-than) and the list concatenation @ (cf. [Pad 94], Section 4). `s` is the successor function for natural numbers.

A point list L represents a rectangle if L comprises the entire interval I between the least and the greatest point of L. It is sufficient to check whether I is a subset of L.

```
rectangle(L) = Sub(interval(minpoint(L))(maxpoint(L)),L)
minpoint([x]) = x
minpoint((x1,y1)::y::L) = (min(x1,x2),min(y1,y2))  <==  minpoint(y::L) = (x2,y2)
maxpoint([x]) = x
maxpoint((x1,y1)::y::L) = (max(x1,x2),max(y1,y2))  <==  maxpoint(y::L) = (x2,y2)
```

`Sub` (set containment) is a standard Boolean function of Expander. `minpoint` and `maxpoint` are defined recursively on nonempty point lists. Most programming languages do not provide set operations like `Sub`. Hence we replace the above definition of `rectangle` by an equivalent one, which does not use `Sub`:

```
rectangle(L) = eq(length(L),((x2-x1)+1)*((y2-y1)+1))
   <== { minpoint(L) = (x1,y1), maxpoint(L) = (x2,y2) }
```

A naive generate-and-test program that computes all dissections of a rectangle R would first apply the partition function `part` of Ex. 2.1.1 to the point list L representing R and then check whether all point lists of each generated partition P of L yield rectangles. If so, P is a dissection into rectangles, otherwise P were discarded. However, this program were only a precise *requirement* specification of the problem. A *design* specification should not only state the problem, but also follow a well-chosen algorithmic paradigm that guides the designer in deriving (efficient) functional code. Programs cannot be constructed automatically from specifications that do not say anything about *how* the specified functions obtain their results or how relational queries get their solutions. Since Expander involves only a few compilation techniques, Expander specifications cannot have the most efficient runtime behaviour. Nevertheless, Expander may serve as a prototyping tool for trying out algorithms by testing them, proving their correctness and relating them to subsequent implementations.

Why would the above-sketched naive generate-and-test program be a bad design? Because only a few partitions of a point list would pass the rectangle test. A better design goal is a program that produces (almost) only partitions that pass the test successfully. Of course, this goal cannot be achieved to its full extent. Often the test candidate must be built up completely before the test can be performed. Even in these cases, however, there should be a sort of pre-test, which allows us to recognize and eliminate non-rectangle partitions in an early stage of their construction.

So the crucial question is: How can the definition of `part` (cf. 2.1.1) be modified such that this function produces only rectangle dissections?

```
rectangles(P) = forall(rectangle)(P)
forall(C)(nil) = true
forall(C)(x::L) = C(x) and forall(C)(L)
```

Both for gaining more general results from this case study and for keeping to the original problems of [Car 93] we add to `rectangles` a variable constraint parameter C and the condition that all admissable partitions have the same given length n. For this purpose `part` is augmented by the parameters C and n. We assume that C holds true for all singleton lists and $n \geq 1$. Hence Axiom 1 of PARTITION is changed into:

```
partC(n,C,[x],[[x]])
```

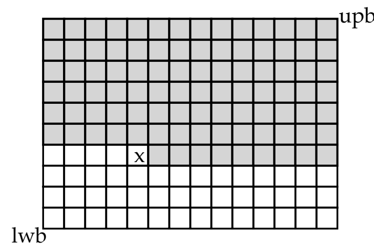

Figure 1.

Partitions of the list x::y::L should be constructed from partitions of the sublist y::L (cf. 2.1.1). If one starts out from the point list returned by a call of `interval`, then x and y::L are located in the interval such as the square x and the gray plane are located in the grid of Fig. 1. By induction hypothesis, a recursive call of `partC` will return a partition P of y::L that pertains to one of the cases of Fig. 2.

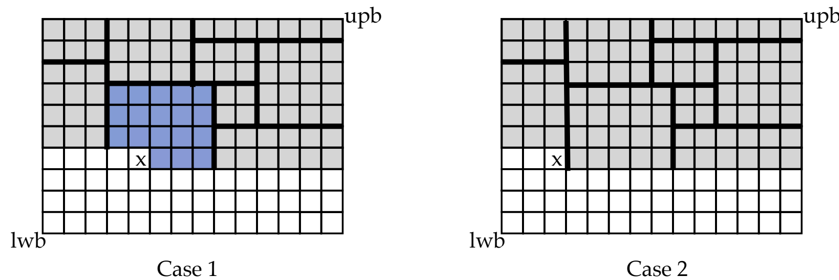

Figure 2.

Since we want to come up with rectangle dissections only, Axiom 2 of PARTITION:

```
part(x::y::L,[x]::P)  <==  part(y::L,P)
```

must be restricted to Case 2. If the singleton [x] were added to P also in Case 1, the dark gray *non*-rectangle would become an element of the partition of the entire interval x::y::L. Hence Axioms 2 and 3 of PARTITION are changed as follows:

```
partC(n,C,x::y::L,P)  <==  { partC(n,C,y::L,Q), glue(n,C,x,Q,P) }
glue(n,C,x,Q,P)  <==  { rectangles(Q) = true, glueAux(n,C,x,Q,P) }
glue(n,C,x,Q,P)  <==  { rectangles(Q) = false, search(nonRect}(C,x),nil,x,Q,P) }
nonRect(C,(x,y))(z::L) = C((x,y)::z::L) and eq((s(x),y),z)
```

`partC` calls the predicate `glue`, which first checks whether all elements of Q are rectangles.

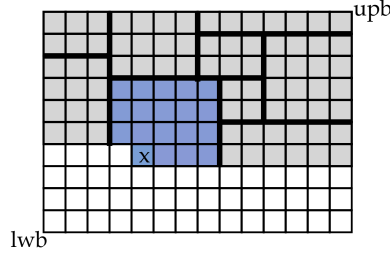

Figure 3. The partition *nonRect*

First, suppose that not all elements of Q are rectangles. Then we are faced with Case 1 above and the predicate `search` is called. There is a single admissable partition *nonRect* of x::y::L, which can be constructed from Q (cf. Fig. 3). For building *nonRect* we have to search for the dark gray plane of Fig. 2. If x = (y,z), this plane is represented by the point list L of Q whose head element is given by (y+1,z). In contrast to the one-dimensional case (cf. Axiom 3 of PARTITION), L is not necessarily the head of Q. The search predicate used above is specified nondeterministically as follows:

```
search(C,Q1,x,L::Q2,(x::L)::(Q1@Q2))  <==  C(L) = true
search(C,Q1,x,L::Q2,P)  <==  search(C,L::Q1,x,Q2,P)
```

`search` returns a partition as a term of the form (x::L)::(Q1@Q2) only if L satisfies the constraint parameter C.

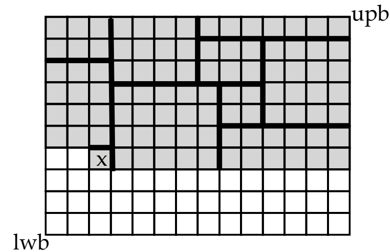

Figure 4.

Second, suppose that all elements of Q are rectangles. Then we are faced with Case 2 of Fig. 2. The predicate `glueAux` is called, which asks whether Q has still less than n elements. If so, [x]::Q is returned as an admissable partition of x::y::L (cf. Fig. 4).

```
glueAux(n,C,x,Q,[x]::Q)  <==  length(Q) < n
```

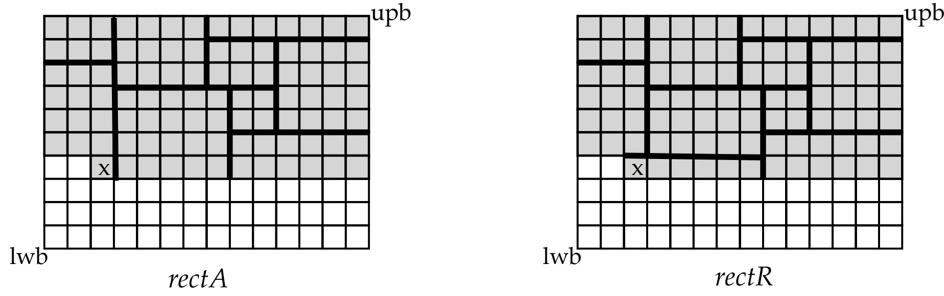

Figure 5. The partitions *rectA* and *rectR*

There are two further admissable partitions of x::y::L that can be constructed from Q, say *rectA* and *rectR* (cf. Fig. 5). Each of them is an extension of a rectangle dissection of y::L. *rectA* is obtained by adding x to the rectangle A above x. *rectR* results from merging x with the rectangle R right to x. *rectR* is admissable only if the rectangle right to x covers a single horizontal line. Hence *rectR* consists of rectangles, while in *rectA*, the rectangle above x becomes a non-rectangle when x is added to it. Using the above-defined predicate `search` we look for A and R. If x = (y,z), A is the first rectangle of Q whose low-right corner is given by (y,z+1), while R is the first line of Q whose leftmost point is given by (y+1,z).

```
glueAux(n,C,x,Q,P)  <==  search(rectAR(C,x),nil,x,Q,P)
rectAR(C,(x,y))(z::L) = C((x,y)::z::L) and (eq((x,s(y)),lowRightCorner(L)) or
                                             (line(L) and eq((s(x),y),z)))
lowRightCorner(L) = (x1,y2)  <==  { last(L) = (x1,y1), hd(L) = (x2,y2) }
line(L) = eq(y1,y2)  <==  { last(L) = (x1,y1), hd(L) = (x2,y2) }
last([x]) = x
last(x::y::L) = last(y::L)
hd(x::L) = x
```

2.2 Expander versus Prolog

All functions and relations explained in the previous section are combined into the following Expander specification:

```
DISSECT
        functs  interval 1  row 2  rectangle 3  minpoint 4 5  maxpoint 6 7
                rectangles 8  forall 9 10 nonRect 19  rectAR 20  incrX 21
                incrY 22  lowRightCorner 23  line 24  hd 25  last 26 27
        preds   partC 11 12  glue 13 14  glueAux 15 16  search 17 18
        vars    x y x1 y1 x2 y2 k n C L P Q Q1 Q2
        axioms
        (1) { interval(x1,y1,x2,y2)
                = ite(gt(y1,y2),nil,row(x1,x2,y1)@interval(x1,s(y1),x2,y2)) }
        (2) { row(x1,x2,y) = ite(gt(x1,x2),nil,(x1,y)::row(s(x1),x2,y)) }
        (3) { rectangle(L) = eq(length(L),((x2-x1)+1)*((y2-y1)+1)) }
                <== { minpoint(L) = (x1,y1), maxpoint(L) = (x2,y2) }
        (4) { minpoint([x]) = x }
```

```
 (5) { minpoint((x1,y1)::y::L) = (min(x1,x2),min(y1,y2)) }
         <== { minpoint(y::L) = (x2,y2) }
 (6) { maxpoint([x]) = x }
 (7) { maxpoint((x1,y1)::y::L) = (max(x1,x2),max(y1,y2)) }
         <== { maxpoint(y::L) = (x2,y2) }
 (8) { rectangles(P) = forall(rectangle)(P) }
 (9) { forall(C)(nil) = true }
(10) { forall(C)(x::L) = C(x) and forall(C)(L) }

(11) { partC(n,C,[x],[[x]]) }
(12) { partC(n,C,x::y::L,P) } <== { partC(n,C,y::L,Q), glue(n,C,x,Q,P) }
(13) { glue(n,C,x,Q,P) } <== { rectangles(Q) = true, glueAux(n,C,x,Q,P) }
(14) { glue(n,C,x,Q,P) } <== { rectangles(Q) = false,
                               search(nonRect(C,x),nil,x,Q,P) }
(15) { glueAux(n,C,x,Q,[x]::Q) } <== { length(Q) < n }
(16) { glueAux(n,C,x,Q,P) } <== { search(rectAR(C,x),nil,x,Q,P) }
(17) { search(C,Q1,x,L::Q2,(x::L)::(Q1@Q2)) } <== { C(L) = true }
(18) { search(C,Q1,x,L::Q2,P) } <== { search(C,L::Q1,x,Q2,P) }

(19) { nonRect(C,x)(L) = C(x::L) and eq(incrX(x),hd(L)) }
(20) { rectAR(C,x)(L) = C(x::L) and (eq(incrY(x),lowRightCorner(L)) or
                                     (line(L) and eq(incrX(x),hd(L)))) }
(21) { incrX(x,y) = (s(x),y) }
(22) { incrY(x,y) = (x,s(y)) }
(23) { lowRightCorner(L) = (x1,y2) } <== { last(L) = (x1,y1), hd(L) = (x2,y2) }
(24) { line(L) = eq(y1,y2) } <== { last(L) = (x1,y1), hd(L) = (x2,y2) }
(25) { hd(x::L) = x }
(26) { last([x]) = x }
(27) { last(x::y::L) = last(y::L) }
```

Axioms 11 to 18 specify our dissection algorithm. The axioms for `partC` and `glue` are unambiguous in the sense that at most one of them is applicable to a given atomic formula. Nevertheless, these relations cannot be defined as functions because they depend on the relations `glueAux` and `search` whose axioms are ambiguous. Hence `partC` and `glue`, which use `glueAux` and `search`, are also multi-valued functions. All other operators of DISSECT are deterministic and are thus defined as single-valued functions.

Even that cannot be maintained if we translate DISSECT into a pure logic program where all functions must be flattened into relations. A function f with n arguments becomes a relation R with $n+1$ parameters such that for all suitable values $a_1,\dots,a_n$, $f(a_1,\dots,a_n)=b$ holds true iff $R(a_1,\dots,a_n,b)$ is valid. A Boolean function f such as `rectangle, rectangles, forall, nonRect, rectAR, line` or `lowRightCorner` can be translated into a relation R more directly:

$$f(a_1,\dots,a_n)=true \quad\Leftrightarrow\quad R(a_1,\dots,a_n).$$

How do we translate an equation $f(a_1,\dots,a_n)=false$? We may use Prolog's definition of negation, apply the cut operator !:

```
not(R) :- R, !, fail.
not(_).
```

and express $f(a_1,\ldots,a_n) = false$ as $not(R(a_1,\ldots,a_n))$. If $f(a_1,\ldots,a_n)$ occurs in a conditional Boolean expression as in the equation $t = ite(f(a_1,\ldots,a_n),u,v)$, it is compiled as follows:

```
t :- R(a1,...,an), !, u.
t :- v.
```

A further adaption to Prolog concerns higher-order functions. The full polymorphism of a higher-order function f like `forall` (cf. DISSECT) cannot be maintained in the logic program. The Prolog definition of f is restricted to the *actual* function parameters occurring in the program. For instance, Axiom 8 of DISSECT is translated as follows:

```
rectangles(P) :- forall(rectangle,P).
forall(_,[]).
forall(C,[X|L]) :- valid(C,X), forall(C,L).
```

Here `rectangle` is just a constant and not a predicate defined in the program. We must define the predicate `valid` such that the actual constant parameter `rectangle` is related back to the predicate `rectangle`:

```
valid(rectangle,L) :- rectangle(L).
```

The other actual parameters of `valid` are handled analogously:

```
valid(nonRect(C,X),L) :- valid(C,[X|L]), incrX(X,I), hd(L,I).
valid(rectAR(C,X),L) :- valid(C,[X|L]), incrY(X,I), lowRightCorner(L,I).
valid(rectAR(C,X),L) :- valid(C,[X|L]), line(L), incrX(X,I), hd(L,I).
```

By introducing cuts for conditionals as proposed above the rest of DISSECT can be translated schematically into Prolog clauses:

```
interval((_,Y1),(_,Y2),[]) :- Y1 > Y2, !.
interval((X1,Y1),(X2,Y2),I) :- row(X1,X2,Y1,R), Y is Y1+1,
                               interval((X1,Y),(X2,Y2),J), append(R,J,I).
row(X1,X2,_,[]) :- X1 > X2, !.
row(X1,X2,Y,[(X1,Y)|R]) :- X is X1+1, row(X,X2,Y,R).
rectangle([]).
rectangle(L) :- minpoint(L,(X1,Y1)), maxpoint(L,(X2,Y2)),
                A is ((X2-X1)+1)*((Y2-Y1)+1), length(L,A).
minpoint([(X1,Y1)],(X1,Y1)) :- !.
minpoint([(X1,Y1)|L],(X,Y)) :- minpoint(L,(X2,Y2)),
                                min(X1,X2,X), min(Y1,Y2,Y).
maxpoint([(X1,Y1)],(X1,Y1)) :- !.
maxpoint([(X1,Y1)|L],(X,Y)) :- maxpoint(L,(X2,Y2)),
                                max(X1,X2,X), max(Y1,Y2,Y).
min(X,Y,X) :- X =< Y, !.
min(_,Y,Y).
max(X,Y,X) :- X >= Y, !.
max(_,Y,Y).

partC(_,_,[X],[[X]]).
```

```
partC(N,C,[X|L],P) :- partC(N,C,L,Q), glue(N,C,X,Q,P).
glue(N,C,X,Q,P) :- rectangles(Q), !, glueAux(N,C,X,Q,P).
glue(_,C,X,Q,P) :- search(nonRect(C,X),[],X,Q,P).
glueAux(N,_,X,Q,[[X]|Q]) :- length(Q,A), A < N.
glueAux(_,C,X,Q,P) :- search(rectAR(C,X),[],X,Q,P).
search(C,Q1,X,[L|Q2],[[X|L]|P]) :- valid(C,[X|L]), append(Q1,Q2,P).
search(C,Q1,X,[L|Q2],P) :- search(C,[L|Q1],X,Q2,P).

incrX((X,Y),(X1,Y)) :- X1 is X+1.
incrY((X,Y),(X,Y1)) :- Y1 is Y+1.
lowRightCorner(L,(X,Y)) :- last(L,(X,_)), hd(L,(_,Y)).
line(L) :- last(L,(_,Y)), hd(L,(_,Y)).
hd([X|_],X).
last([X],X) :- !.
last([_|L],X) :- last(L,X).
```

2.3 Tests and proofs

Conjecture 1 of the following extension of PARTITION and DISSECT (cf. 2.1.1 and 2.2) provides a test case, while Conjectures 2 and 3 state the correctness of `partC` and `part`, respectively. Two auxiliary functions occur in these conjectures: `flatten` concatenates the elements of a list of lists into a single list. `isPartOf(P,L)` checks whether the flattened version of `P` has the same elements as `L`.

```
DISSECTcheck
        base     PARTITION  DISSECT
        functs   isPartOf 1  flatten 2 3  >> 4
        vars     I P'
        axioms
        (1)      { isPartOf(P,L) = eq(set(flatten(P)),set(L)) }
        (2)      { flatten(nil) = nil }
        (3)      { flatten(L::P) = L@flatten(P) }
        (4)      { (x::L,P) >> (L,P') = true }
        theorems
        (1)      { x::L = [x]@L }
        (2)      { x::(L@L') = (x::L)@L' }
        conjects
        (1)      { I = interval(1,1)(3,3), partC(5,fn(L)(leq(length(L),2)),I,P),
                   b = leq(length(P),5) and rectangles(P) and isPartOf(P,I) }
        (2)      { leq(length(P),n) and rectangles(P) and isPartOf(P,I) = true }
                    <== { partC(n,C,L,P) }
        (3)      { L = flatten(P) } <== { part(L,P) }
```

`set` is a standard function, which makes a list into a set. Consequently, the equation $set(L_1) = set(L_2)$ holds true iff L_1 and L_2 have the same elements. `fn(L)(e)` denotes the function defined by the λ-expression $\lambda L.e$.

Here is part of the Expander output that results from requesting a solution of Conjecture 1:

```
initial conclusion:
 (1) { I = interval(1,1)(3,3), partC(5,fn(L)(leq(length(L),2)),I,P),
```

```
      b = leq(length(P),5) and (rectangles(P) and isPartOf(P,I)) }
conclusion:
 (1) { I = [(1,1),(2,1),(3,1),(1,2),(2,2),(3,2),(1,3),(2,3),(3,3)],
      P = [[(1,1)],[(2,1),(3,1)],[(1,2),(1,3)],[(2,2),(3,2)],[(2,3),(3,3)]],
      b = true }
 (2) { ... }
 (3) { ... }
 (4) { ... }
 (5) { ... }
solved goals: 1
narrowed goals have been deleted
```

The *conclusion* implies the *initial conclusion* from which it is derived by *backward resolution* and *paramodulation* (cf. [Pad 95]). Each conclusion and each *premise* (see below) is a *goal set*, i.e. a disjunction of goals where a goal is a conjunction (written as a numbered set) of atomic formulas. Goal 1 of the above conclusion represents a solution. Further solutions are derivable from Goals 2 to 5.

Expander does not only support tests of a specification. Conjectures usually represent correctness conditions on functions or relations. While the above solution of Conjecture 1 only implies that the first partition generated by `partC` fulfils all requirements, Conjecture 2 means that all partitions produced by `partC` satisfy the requirements. Expander provides inference rules for constructing proofs of such correctness conditions. Instead of presenting the long verification of `partC` (Conjecture 2), we restrict ourselves to the correctness of `part` (Conjecture 3). Since the fundamental algorithmic idea of `part` is also involved in `partC`, Conjecture 3 captures a basic aspect of Conjecture 2.

Using Expander, a proof of a *Gentzen clause* $gs \Leftarrow hs$ such as Conjectures 2 and 3 is carried out interactively. One starts with singleton lists $front = [gs]$ and $rear = [hs]$ and extends them stepwise into lists

$$front = [gs, gs_1, \ldots, gs_k] \quad \text{and} \quad rear = [hs, hs_1, \ldots, hs_n],$$

consisting of successively inferred goal sets. *front* is a backward proof of the clause $gs \Leftarrow gs_k$ consisting of backward resolution and paramodulation steps (see above). *rear* is a forward proof of $hs_n \Leftarrow hs$ consisting of *forward resolution* and *paramodulation* steps (cf. [Pad 95]). The proof of $gs \Leftarrow hs$ is complete if there are $k, n \geq 1$ such that hs_n *subsumes* gs_k. This syntactical condition implies that $gs_k \Leftarrow hs_n$ is inductively valid (cf. [Pad 94], Section 3.5). From the validity of $gs \Leftarrow gs_k$, $gs_k \Leftarrow hs_n$ and $hs_n \Leftarrow hs$ one concludes that the original conjecture $gs \Leftarrow hs$ holds true.

The (inductive) proof of Conjecture 3 uses Theorems 1 and 2 and the Boolean function >> as lemmas and the induction ordering, respectively:

```
initial conclusion:
 (1) { L = flatten(P) }
initial premise:
 (1) { part(L,P) }

atom 1 in premise goal 1 replaced with axiom PARTITION1
atom 1 in premise goal 1 replaced with axiom PARTITION2
atom 1 in premise goal 1 replaced with axiom PARTITION3
premise:
 (1) { L = x::(y::L1), P = (x::L')::P1, part(y::L1,L'::P1) }
 (2) { L = x::(y::L1), P = ([x])::P1, part(y::L1,P1) }
```

```
 (3) { L = [x], P = [[x]] }

atom 3 in premise goal 1 replaced with conjecture 3
premise:
 (1) { (L,P)>>(y::L1,L'::P1) = false, L = x::(y::L1), P = (x::L')::P1,  part(y::L1,L'::P1) }
 (2) { y::L1 = flatten(L'::P1), L = x::(y::L1), P = (x::L')::P1 }
 (3) { L = x::(y::L1), P = ([x])::P1, part(y::L1,P1) }
 (4) { L = [x], P = [[x]] }

term at position 1 1 in premise goal 1 replaced with axiom DISSECTcheck4
premise:
 (1) { y::L1 = flatten(L'::P1), L = x::(y::L1), P = (x::L')::P1 }
 (2) { L = x::(y::L1), P = ([x])::P1, part(y::L1,P1) }
 (3) { L = [x], P = [[x]] }

atom 3 in premise goal 2 replaced with conjecture 3
premise:
 (1) { y::L1 = flatten(L'::P1), L = x::(y::L1), P = (x::L')::P1 }
 (2) { (L,P)>>(y::L1,P1) = false, L = x::(y::L1), P = ([x])::P1, part(y::L1,P1) }
 (3) { y::L1 = flatten(P1), L = x::(y::L1), P = ([x])::P1 }
 (4) { L = [x], P = [[x]] }

term at position 1 1 in premise goal 2 replaced with axiom DISSECTcheck4
premise:
 (1) { y::L1 = flatten(L'::P1), L = x::(y::L1), P = (x::L')::P1 }
 (2) { y::L1 = flatten(P1), L = x::(y::L1), P = ([x])::P1 }
 (3) { L = [x], P = [[x]] }

term at position 2 2 2 replaced with equation 1 in premise goal 1
premise:
 (1) { y::L1 = flatten(L'::P1), x::flatten(L'::P1) = L, P = (x::L')::P1 }
 (2) { y::L1 = flatten(P1), L = x::(y::L1), P = ([x])::P1 }
 (3) { L = [x], P = [[x]] }

term at position 2 2 2 replaced with equation 1 in premise goal 2
premise:
 (1) { y::L1 = flatten(L'::P1), x::flatten(L'::P1) = L, P = (x::L')::P1 }
 (2) { y::L1 = flatten(P1), x::flatten(P1) = L, P = ([x])::P1 }
 (3) { L = [x], P = [[x]] }

term at position 2 1 2 in premise goal 1 replaced with axiom DISSECTcheck3
premise:
 (1) { x::(L'@flatten(P1)) = L, y::L1 = flatten(L'::P1), P = (x::L')::P1 }
 (2) { y::L1 = flatten(P1), x::flatten(P1) = L, P = ([x])::P1 }
 (3) { L = [x], P = [[x]] }

term at position 1 1 in premise goal 1 replaced with theorem 2
premise:
```

```
 (1) { (x::L')@flatten(P1) = L, y::L1 = flatten(L'::P1), P = (x::L')::P1 }
 (2) { y::L1 = flatten(P1), x::flatten(P1) = L, P = ([x])::P1 }
 (3) { L = [x], P = [[x]] }

term at position 2 1 in premise goal 2 replaced with theorem 1
premise:
 (1) { (x::L')@flatten(P1) = L, y::L1 = flatten(L'::P1), P = (x::L')::P1 }
 (2) { ([x])@flatten(P1) = L, y::L1 = flatten(P1), P = ([x])::P1 }
 (3) { L = [x], P = [[x]] }

term at position 1 2 in conclusion goal 1 replaced with axiom DISSECTcheck2
term at position 1 2 in conclusion goal 1 replaced with axiom DISSECTcheck3
conclusion:
 (1) { P = L1::P1, L1@flatten(P1) = L }
 (2) { P = nil, L = nil }

term at position 2 1 2 in conclusion goal 1 replaced with axiom DISSECTcheck2
term at position 2 1 2 in conclusion goal 1 replaced with axiom DISSECTcheck3
conclusion:
 (1) { P = L1::P1, L1@flatten(P1) = L }
 (2) { L1@(L2@flatten(P1)) = L, P = L1::(L2::P1) }
 (3) { P = [L] }
 (4) { P = nil, L = nil }

term at position 2 2 1 replaced with equation 1 in premise goal 3
premise:
 (1) { (x::L')@flatten(P1) = L, y::L1 = flatten(L'::P1), P = (x::L')::P1 }
 (2) { ([x])@flatten(P1) = L, y::L1 = flatten(P1), P = ([x])::P1 }
 (3) { L = [x], [L] = P }

conjecture 3 has been proved
```

2.4 Dimensioning dissections

[Car 93] handles three ways of dimensioning dissections: *dimensioning by area, dimensioning by brick size* and *dimensioning by proportion*. In all three cases the generated dissections consist of the same number n of rectangles. For dimensioning by area, n must divide $x*y$ and the quotient a is the area covered by each rectangle of the dissection. For dimensioning by brick size, a is passed as a parameter, n is defined as $\lceil (x*y)/a \rceil$ and each rectangle of the dissection covers at most a points.[3]. In both cases the desired relationship between n and a is achieved by setting the constraint C on each generated point list L to: $length(L) \leq a$ (cf. 2.1).

Although **glueAux** takes n only as an upper bound (cf. Axiom 15 of DISSECT), all dissections returned by **partC** consist of exactly $k = n$ rectangles. Others would lead to a contradiction: If $k < n$, then, in the case of dimensioning by area, $a = (x*y)/n$ would entail

$$x*y \leq k*a = k*((x*y)/n) < n*((x*y)/n) = x*y,$$

[3] For the sake of simplicity and since we restrict ourselves to integer arithmetic, these definitions differ slightly from [Car93]

and, in the case of dimensioning by brick size, $n = \lceil (x * y)/a \rceil$ would imply

$$x * y + r \leq k * a + r < n * a = \lceil (x * y)/a \rceil * a \leq x * y + r$$

for some $r < a$.

For dimensioning by proportion, both n and a length-to-height ratio p for each rectangle are passed as parameters. The constraint C on generated point lists is set to *true* because each dissection can be checked for admissability only after its construction has been completed.

```
DISSECT_E
        base    DISSECT
        functs  makeRect 1  leqArea 2  proportioned 3
        preds   areaDissect 4  brickDissect 5  propDissect 6
        vars    a f p l h I
        axioms
        (1)     { makeRect(L) = rect(x1,y1,x2-x1,y2-y1) }
                    <== { (s(x1),s(y1)) = minpoint(L), (x2,y2) = maxpoint(L) }
        (2)     { leqArea(a)(L) = leq(length(L),a) }
        (3)     { proportioned(p)(L) = eq(p*l,h) or eq(p*h,l) }
                    <== { makeRect(L) = rect(x,y,l,h)) }
        (4)     { areaDissect(x,y,n,P) }
                    <== { I = interval(1,1)(x,y), a = (x*y)/n, partC(n,leqArea(a),I,P) }
        (5)     { brickDissect(x,y,a,P) }
                    <== { I = interval(1,1)(x,y), n = (x*y)//a, partC(n,leqArea(a),I,P) }
        (6)     { propDissect(x,y,n,p,P) }
                    <== { I = interval(1,1)(x,y), partC(n,fn(L)(true),I,P),
                          length(P) = n, forall(proportioned(p))(P) = true }
```

// denotes division followed by the ceiling operator $\lceil - \rceil$. `makeRect` transforms a point list L into a term of the form *rect(x,y,l,h)* where $(x + 1, y + 1)$ is the least point, l is the length and h is the height of the rectangle represented by L. DISSECT_E is translated as follows into an extension of the Prolog program given in Section 2.2:

```
makeRect(L,rect(X,Y,Length,Height)) :- minpoint(L,(X1,Y1)),
                                       maxpoint(L,(X2,Y2)),
                                       X is X1-1, Y is Y1-1,
                                       Length is X2-X, Height is Y2-Y.

areaDissect(X,Y,N,P) :- interval((1,1),(X,Y),I), A is (X*Y)//N, partC(N,leq(A),I,P).
brickDissect(X,Y,A,P) :- interval((1,1),(X,Y),I), D is X*Y, divC(D,A,N),
                         partC(N,leq(A),I,P).
divC(X,Y,Q) :- R is X mod Y, R = 0, !, Q is X//Y.
divC(X,Y,Q) :- Q is (X//Y)+1.
propDissect(X,Y,N,A,P) :- interval((1,1),(X,Y),I), partC(N,true,I,P),
                          length(P,N), forall(proportioned(A),P).

valid(true,_).
valid(leq(A),L) :- length(L,Length), Length =< A.
valid(proportioned(A),L) :- makeRect(L,rect(X,Y,Length,Height)),
```

```
                                   (Height is A*Length; Length is A*Height).
```

Again we had to replace actual function parameters (`true`, `leq(A)` and `proportioned(A)`) of a higher-order function (`valid`) by homonymous constants. When the above program is combined with the one given in Section 2.2, the clauses for `valid` must be grouped together. Otherwise Prolog treats them as separate definitions.

3 Deriving functional code

3.1 Computing dissections

The deterministic parts of our dissection algorithm, i.e. Axioms 1 to 10 and 19 to 27 of DISSECT (cf. 2.2), are turned directly into ML function definitions. In particular, the premises of Axioms 3, 5, 7, 23 and 24 become conditional expressions or local definitions.

```
fun interval(x1,y1,x2,y2) = if y1 > y2 then nil
                                       else row(x1,x2,y1)@interval(x1,y1+1,x2,y2)
and row(x1,x2,y) = if x1 > x2 then nil else (x1,y)::row(x1+1,x2,y)

fun rectangle(L) = let val (x1,y1) = minpoint(L)
                       val (x2,y2) = maxpoint(L)
                   in length(L) = ((x2-x1)+1)*((y2-y1)+1) end
and minpoint[x] = x
|   minpoint((x1,y1)::L) = let val (x2,y2) = minpoint(L)
                           in (min(x1,x2),min(y1,y2)) end
and maxpoint[x] = x
|   maxpoint((x1,y1)::L) = let val (x2,y2) = maxpoint(L)
                           in (max(x1,x2),max(y1,y2)) end

fun rectangles(P) = forall(rectangle)(P)

and forall _ (nil) = true
|   forall(C)(x::L) = C(x) andalso forall(C)(L)

fun nonRect(C,x)(L) = C(x::L) andalso incrX(x) = hd(L)
and rectAR(C,x)(L) = C(x::L) andalso (incrY(x) = lowRightCorner(L) orelse
                                      (line(L) andalso incrX(x) = hd(L)))
and incrX(x,y) = (x+1,y)
and incrY(x,y) = (x,y+1)
and lowRightCorner(L) = let val (x,_) = last(L)
                            val (_,y) = hd(L) in (x,y) end
and line(L) = let val (x,_) = last(L)
                  val (y,_) = hd(L) in x = y end
and last[x] = x
|   last(x::L) = last(L)
```

It remains to translate the relations `partC`, `glue`, `glueAux` and `search` (cf. 2.1). Each of them is used with a fixed input-output mode. The last argument takes the values of the associated multi-valued function. Hence

we proceed as in Examples 1.5 and 1.6 and store these values into streams.[4] We recapitulate Axioms 11 to 18 of DISSECT, which define the above relations (cf. 2.1):

```
(11) { partC(n,C,[x],[[x]]) }
(12) { partC(n,C,x::y::L,P) } <== { partC(n,C,y::L,Q), glue(n,C,x,Q,P) }
(13) { glue(n,C,x,Q,P) } <== { rectangles(Q) = true, glueAux(n,C,x,Q,P) }
(14) { glue(n,C,x,Q,P) } <== { rectangles(Q) = false,
                               search(nonRect(C,x),nil,x,Q,P) }
(15) { glueAux(n,C,x,Q,[x]::Q) } <== { length(Q) < n }
(16) { glueAux(n,C,x,Q,P) } <== { search(rectAR(C,x),nil,x,Q,P) }
(17) { search(C,Q1,x,L::Q2,(x::L)::(Q1@Q2)) } <== { C(L) = true }
(18) { search(C,Q1,x,L::Q2,P) } <== { search(C,L::Q1,x,Q2,P) }
```

When compiling multi-valued functions, i.e. relations with a fixed input-output mode, into single-valued stream functions, we distinguish between *linear* and *nonlinear* nondeterminism. The above definition of `partC` is nonlinearly nondeterministic because an expansion of Axiom 12 iteratively splits into several branches, one for each value of `Q` that is produced by a recursive call of `partC` and then passed over to `glue`. Such an "expansion tree" must be linearized by an equivalent stream function. Hence we need the higher-order function `MapS` (cf. Section 1.2), which first applies a multi-valued function to stream elements and then concatenates the elements of the resulting stream of streams into a single stream. With the help of `MapS` Axioms 11 and 12 are translated into the following function definition:

```
fun partC(n,C,[x]) = [[x]] & (fn()=>Nil)
|   partC(n,C,x::L) = MapS(glue(n,C,x))(partC(n,C,L))
```

Each recursive call of `partC` generates a stream each of whose elements is mapped by `glue(n,C,x)` to a substream. `MapS` "flattens" all substreams into a single stream.

The definitions of `glue` and `glueAux` and `search`, given by Axioms 13 to 18, are linearly nondeterministic because each of them involves only one recursive call of a multi-valued function. Here we distinguish between *ambiguously* defined functions like `glueAux` and `search` (cf. Axioms 15 to 18) and *unambiguously* defined functions like `glue` (cf. Axioms 13 and 14). The latter can be translated easily with the help of a conditional:

```
and glue(n,C,x)(Q) = if rectangles(Q) then glueAux(n,C,x,Q)
                                      else search(1, nonRect(C,x),nil,x,Q)
```

Ambiguous definitions, on the other hand, include several axioms that pertain to the same function call. Expressed in terms of the underlying inference mechanism, the same call *resolves* upon several axioms. Ambiguous definitions are the main feature of a logic language for implementing nondeterminism. They provide the choice points where the above-mentioned expansion tree branches out and leads to several values of the same call. In a functional language an ambiguous definition must be splitted into *disjoint* cases such that, for each argument, the stream of all corresponding values is returned even if they are distributed over several axioms of the specification.

```
and glueAux(n,C,x,Q) = if length(Q) < n
                       then ([x]::Q)&(fn()=>search(2,rectAR(C,x),nil,x,Q))
                       else search(2,rectAR(C,x),nil,x,Q)
```

[4][LPP93] and [Pad93] present another case study where programming with streams plays a crucial role.

```
and search(2,C,Q1,x,L::Q2) = if C(L)
                              then ((x::L)::(Q1@Q2))&(fn()=>search(1,C,L::Q1,x,Q2))
                              else search(2,C,L::Q1,x,Q2)
|   search(1,C,Q1,x,L::Q2) = if C(L) then ((x::L)::(Q1@Q2))&(fn()=>Nil)
                                      else search(1,C,L::Q1,x,Q2)
|   search _ = Nil
```

The first parameter of **search** indicates how many values the respective call of this function has to deliver.

DISSECT_E (cf. 2.4) also falls into a deterministic part (Axioms 1 to 3), which can be translated directly into ML function definitions, and multi-valued function definitions (Axioms 4 to 6), which must be simulated by stream functions. Expander's standard constructor *rect* corresponds to an ML datatype:

```
datatype Rect = rect of int*int*int*int
```

Axioms 1 to 3 are compiled as follows:

```
fun makeRect(L) = let val (x1,y1) = minpoint(L)
                      val (x2,y2) = maxpoint(L)
                      val (x,y) = (x1-1,y1-1)
                  in rect(x,y,x2-x,y2-y) end
fun leqArea(a)(L) = length(L) <= a
fun proportioned(p)(L) = let val rect(x,y,l,h) = makeRect(L)
                         in p*l = h orelse p*h = l end
```

With the help of the stream function `Filter` (cf. 1.2) the translation of Axioms 4 to 6 is straightforward:

```
fun areaDissect(x,y,n) = let val I = interval(1,1,x,y)
                             val a = (x*y) div n
                             val s = partC(n,leqArea(a),I)
                         in (s,x+1,y+1) end

fun brickDissect(x,y,a) = let val I = interval(1,1,x,y)
                              val n = ceiling(real(x*y)/real(a))
                              val s = partC(n,leqArea(a),I)
                          in (s,x+1,y+1) end

fun propDissect(x,y,n,p) = let val I = interval(1,1,x,y)
                               val s = partC(n,fn(L)=>true,I)
                               fun C(P) = length(P)=n andalso
                                          forall(proportioned(p))(P)
                               val s = Filter(C)(s)
                           in (s,x+1,y+1) end
```

Since we need to know the size $(x+1, y+1)$ of the plane to be dissected when printing a dissection, `areaDissect`, `brickDissect` and `propDissect` return this value in addition to the stream expression s. The elements of s are not yet constructed when one of the above functions is called. Dissections are computed only when the procedure `next` is called, which shifts the stream pointer `s_ptr` to further elements of s. The structure of `next` is as follows. More details are given in Section 3.2.

```
val s_ptr = ref(Nil:(int*int) list list stream)
val dissSize = ref(0,0)
val boxPos = ref(0,0)
val file = ref(std_out)
fun next(0) = finish()
|   next(n) = case !s_ptr of Nil => finish()
                           | P&s => ... s_ptr:= s(); next(n-1) end
and finish() = (file:= open_out"DISSECT.eps";...;close_out(!file))
```

3.2 Drawing dissections

Expander provides a simple ASCII output of rectangle dissections. This may be sufficient for testing and verifying the specification. For running the functional program a nicer output is desirable. To this end we define a simple compiler of dissection streams into PostScript code:

```
val maxlen = 35 and space = 15
fun start(h,code) = "%!PS-Adobe-3.0 EPSF-3.0\n"^
                    "%%BoundingBox: 5 5 500 "^
                    makestring(h*space+5)^"\n"^
                    "2 setlinewidth 1 setlinejoin\n"^
                    makestring(space)^" "^makestring(space)^" translate\n"^
                    code^"showpage"

fun draw(nil) = ""
|   draw(rect(x,y,l,h)::rs) = let val numbers = fold(widen)[x,y,l,h]""
                              in "0.7 setgray "^numbers^"rectfill\n"^
                                 "0 setgray "^numbers^"rectstroke\n"^
                                 draw(rs) end

and widen(x:int,str) = if x >= 0 then makestring(x*space)^" "^str
                                 else "-"^makestring(~x*space)^" "^str

fun translate(l,h,x) = let val (l,h) = if x+l+l <= maxlen then (l,0) else (l-x,h)
                       in widen(l,widen(h,"translate\n")) end
```

All these functions compute PostScript code, which is assembled as soon as the stream pointer `s_ptr` is shifted by calling `next` (cf. 3.1). For this purpose `next` is equipped with suitable calls of `start`, `draw` and `translate`:

```
val code = ref""
fun init(s) = let val (s,x,y) = s
              in s_ptr:= s; dissSize:= (x,y); boxPos:= (x,y); code:= "" end
fun next(0) = finish()
|   next(n) = case !s_ptr of Nil => finish()
                           | P&s => let val (l,h) = !dissSize
                                        val (x,y) = !boxPos
                                        val (x1,y1) = (x+l,y+h)
                                        val rs = map(makeRect)(P)
```

```
                                            in code:= !code^draw(rs)^translate(l,h,x);
                                               boxPos:= (if x1+l <= maxlen then (x1,y)
                                                          else (l,y1));
                                               s_ptr:= s(); next(n-1) end
      and finish() = (file:= open_out"DISSECT.eps";
                      output(!file,start(#2(!boxPos),!code));close_out(!file))
```

3.3 Sample dissection streams

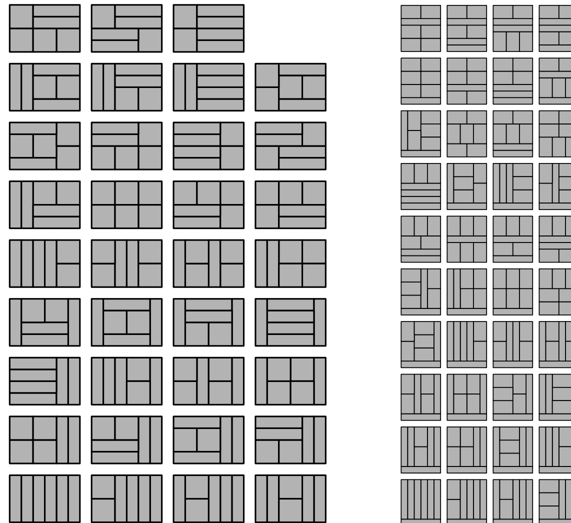

Figure 6. `init(areaDissect(6,4,6)); next(40); init(areaDissect(6,7,7)); next(40);`

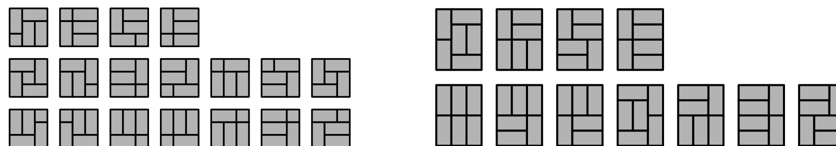

Figure 7. `init(brickDissect(3,3,2)); next(20); init(propDissect(3,4,6,2)); next(20);`

4 A shape grammar solution

The dissection algorithm `partC` presented in 2.1 can be regarded as a specialization of `part` (cf. 2.1.1) to 2-dimensional intervals. The core of the algorithm is given by Axioms 11 to 18 of the specification DISSECT (cf. 2.2). Since the basic algorithmic idea of `partC` is already involved in `part`, we could make plausible the correctness of `partC` by presenting a correctness proof of `part` (cf. 2.3). Furthermore, the translation of DISSECT into functional code was easy because only Axioms 11 and 12 define a nonlinearly nondeterministic function, while the rest of DISSECT specifies deterministic or linearly nondeterministic functions (cf. 3.1). The performance of the program is quite good, although its execution time increases with the size of the plane to be dissected and the number of admissable dissections. Large unevaluated stream expressions lead to a space problem. Although much effort is currently invested into efficient compilers for functional languages, the storage management does not yet solve all the space problems caused by the free use of functional types. So far the only way to achieve greater efficiency is to think about alternative algorithms.

4.1 The specification

To this end we review Carlson's paper [Car 93] that initiated our treatment of the whole subject. His dissection algorithm is based on a *shape grammar*, which defines transitions between particular dissection states. When implementing the rules in his logic language Grammatica (cf. 1.1) Carlson seems to accept a number of ambiguities as to which rules are applicable to which rectangles of a given dissection. Moreover, dissections seem to be regarded as *sets* rather than *lists* of rectangles because two of the grammar rules transform several, not necessarily adjacent, rectangles simultaneously.

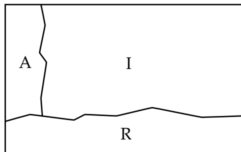

Figure 8. The structure of a dissection state

A closer look at the grammar reveals that certain ambiguities can be avoided if a dissection state is not defined as the set D of all rectangles comprising a dissection, but as a triple (R, A, I) of disjoint rectangle sets. R consists of all rectangles adjacent to the bottom line of the plane to be dissected, A consists of all rectangles adjacent to at the left line (except those of R) and I consists of the rest, i.e. all *inner rectangles*: $I = D-(R \cup A)$ (cf. Fig. 8). Based on these dissection states the grammar reads as in Fig. 9.

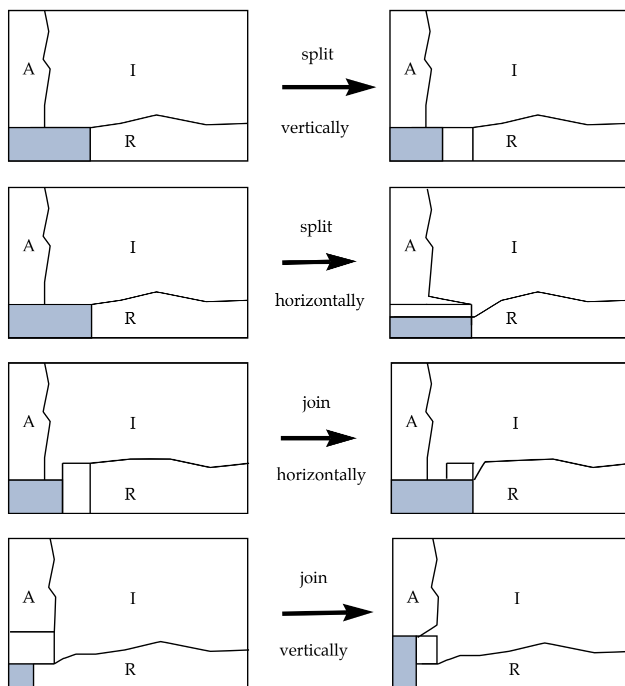

Figure 9. A shape grammar for generating dissections

The gray rectangle denotes the head element r of R. *split vertically* divides r into two rectangles r_1 and r_2, which become the first two elements of R. *split horizontally* divides r into two rectangles r_1 and r_2 such that r_1 becomes the head element of R and r_2 becomes the head element of A. *join horizontally* merges r with a part of the second element r' of R such that the rest of r' becomes a new element of I. *join vertically* merges the r with a part of the head element a of A such that the rest of a becomes a further element of I.

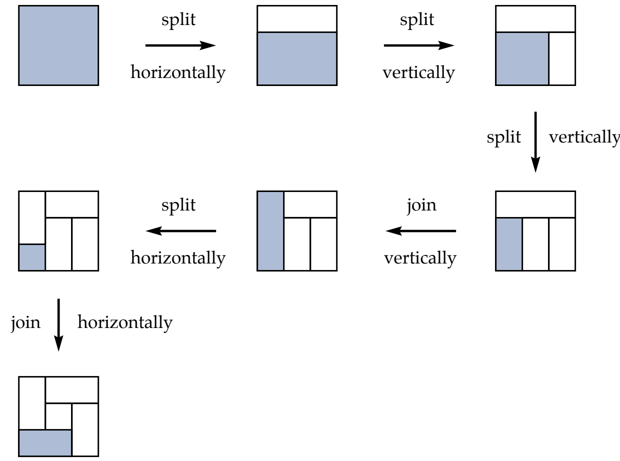

Figure 10. A derivation of a dissection

Example 4.1.1 The transition sequence of Fig. 10 leads to a solution of the query *brickDissect(3,3,2,P)* (cf. 2.4). Starting out from the initial state, given by a one-element dissection, the final state, given by a five-element dissection, is obtained after six rule applications. ▯

Note that the shape grammar of Fig. 9 is ambiguous: the same dissection may be obtained from different transition sequences. Hence, in contrast to the dissection algorithm of 2.1, identical dissections may occur in the set of dissections derived by the grammar. Termination is obvious for the algorithm of Section 2.1. The new algorithm terminates because each grammar rule decreases the state (R, A, I) it is applied to with respect to the following well-founded ordering $>>$. Let a be the area of the original plane.

$$(R, A, I) >> (R', A', I') \quad \Longleftrightarrow_{def} \quad (a - length(I), a - length(R@A)) > (a - length(I'), a - length(R'@A')).$$

Split steps increase R or A, but leave I unchanged. A join step decreases R or A and increases I by one element. Completeness in the sense that all dissections of a given rectangle are computed is also obvious for the algorithm of Section 2.1. r, the head element of R (see above), has one of the neighbourhoods of Fig. 11. The

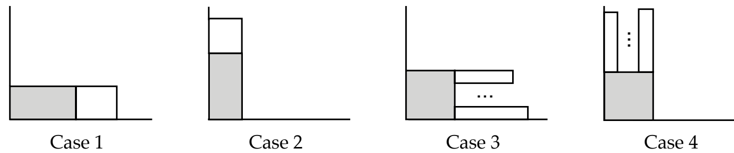

Figure 11.

last step of a derivation into a state of Case 1 or 2 (Case 3 or 4) is a split (join) step. Hence the preceding state is greater w.r.t. $>>$. Since $>>$ has not only a lower, but also an upper bound, namely (a, a), we conclude by induction hypothesis that all dissections of a given plane are derivable with the grammar of Fig. 9.

In contrast to the rectangles of R or A an inner rectangle i is never changed once it has been generated. Hence i can be checked for a given constraint C immediately after i has been created. The number of possible split rule applications to a state (R, A, I) depends on the area of the head element r of R. If r covers more than two points, then (R, A, I) has at least two split rule redices. However, each state admits at most one join rule application. For applying *join horizontally* the height of the second element of R must exceed the height of r. For applying *join vertically* the length of the head element of A must exceed the length of r. Look at the two join rules and you will see that their applicability conditions cannot hold simultaneously. Hence we specify the

split rules as a relation on states and the join rules as a function from states to states.

```
split((r::R,A,I), (rect(0,0,l',h)::rect(l',0,s(s(l))-l',h)::R,A,I))
   <==  { r = rect(0,0,s(s(l)),h), l' isIn (1,s(l)) }

split((r::R,A,I), (rect(0,0,l,h')::R,rect(0,h',l,s(s(h))-h')::A,I))
   <==  { r = rect(0,0,l,s(s(h))), h' isIn (1,s(h)) }

x isIn (x,z)
x isIn (y,z) <== { y < z, x isIn (s(y),z) }

join(rect(0,0,l,h)::rect(x,0,l',h')::R,A,I)
  = (rect(0,0,l+l',h)::R,A,rect(x,h,l',h'-h)::I)  <==  h' > h

join(rect(0,0,l,h)::R,rect(0,y,l',h')::A,I)
  = (rect(0,0,l,h+h')::R,A,rect(l,y,l'-l,h')::I)  <==  l' > l
```

In other words, `split` and `join` define the transition steps of a finite automaton with an initial state of the form $(R, A, I) = ([r], nil, nil)$. The following relation `trans` computes the transitive closure of `split` and `join`:

```
trans(C,(R,A,I),0,0,(R,A,I))  <==  forall(C)(R@A) = true
trans(C,state1,s(k),l,state2)  <==  {split(state1,state), trans(C,state,k,l,state2)}
trans(C,state1,k,s(l),state2)  <== {join(state1) = state, state = (R,A,inner::I),
                                     C(inner) = true, trans(C,state,k,l,state2)}
```

trans(C,state1,k,n,state2) is valid if successive split and join steps lead from *state1* to *state2*. Each inner rectangle created by a join step is checked for satisfying the constraint parameter C. The step is completed only if C holds true. Since inner rectangles are never changed once they have been generated, `trans` checks them immediately after their creation. The admissability of the rectangles of R and A, however, can only be decided at the end of a transition sequence.

k and l are the numbers of split and join steps, respectively, that build up a transition sequence. The number of split steps to decompose a plane of size $x \times y$ into a dissection D consisting of n rectangles is exactly $n - 1$. The number of join steps is at least $n - (x + y - 1)$[5] and at most $n - 2$. These bounds follow from the fact that, for the final state (R, A, I) with $R@A@I = D$, $length(I) = length(D) - length(R@A)$ coincides with the number of join steps to build D, R may have at least one and at most x elements, A may have at least one and at most $y - 1$ elements and thus

$$n - (x + y - 1) \leq length(D) - length(R@A) \leq n - 2.$$

Hence we set $k = n - 1$ and and choose l among $(n + 1) - (x + y), \ldots, n - 2$ when applying `trans` to an initial state $([r], nil, nil)$ (see above):

```
dissect(C,x,y,n,state)  <==  { r = rect(0,0,x,y), l isIn ((n+1)-(x+y),n-2),
                                trans(C,([r],nil,nil),n-1,l,state) }

forall(C)(nil) = true
forall(C)(x::L) = C(x) and forall(C)(L)
```

[5] "-" denotes natural number substraction.

dissect(C,x,y,n,(R,A,I)) holds true if $R@A@I$ is a dissection of *rect(0,0,x,y)* into n rectangles, which satisfy C. The three ways of dimensioning dissections treated in Section 2.4 are turned into calls of `dissect`:

```
areaDissect(x,y,n,states)  <==  { a = (x*y)/n, dissect(eqArea(a),x,y,n,states) }
eqArea(a)(rect(x,y,l,h)) = eq(l*h,a)
brickDissect(x,y,a,states)  <==  { n = (x*y)//a, dissect(leqArea(a),x,y,n,states) }
leqArea(a)(rect(x,y,l,h)) = leq(l*h,a)
propDissect(x,y,n,p,states)  <==  dissect(proportioned(p),x,y,n,states)
proportioned(p)(rect(x,y,l,h)) = eq(p*l,h) or eq(p*h,l)
```

The translation of the above specification into Prolog is quite similar to the translation of DISSECT_E (cf. Sections 2.2 and 2.4). Details are left to the reader.

4.2 The functional program

The main relation `partC` of our dissection algorithm is nonlinearly nondeterministic (cf. 3.1). The same holds true for `trans` and thus for `dissect`. However, in contrast to the axioms for `partC` (cf. 2.1), the three axioms of `trans` (cf. 4.1) yield an ambiguous definition, like the axioms for `glueAux` and `search` (cf. 3.1). Hence the definition of `trans` must be splitted into disjoint cases such that, for each argument of `trans`, the stream of all corresponding values is returned even if they are distributed over several axioms of the specification. Besides `trans` we introduce two auxiliary functions `splitAndTrans` and `joinAndTrans`:

```
exception Join

fun trans(C,0,0)(R,A,I) = if forall(C)(R@A) then (R,A,I) & (fn()=>Nil) else Nil
|   trans(C,k,l)(state) = splitAndTrans(C,k,l,state) %
                          (fn()=>joinAndTrans(C,k,l,state) handle Join => Nil)

and splitAndTrans(_,0,_,_) = Nil
|   splitAndTrans(C,k,l,state) = MapS(trans(C,k-1,l))(splitV(state) % (fn()=>splitH(state)))

and joinAndTrans(_,_,0,_) = Nil
|   joinAndTrans(C,k,l,state) = let val state = join(state)
                                    val (_,_,inner::_) = state
                                in if C(inner) then trans(C,k,l-1)(state) else Nil end
```

`splitAndTrans` and `joinAndTrans` compute streams of states, starting with the successor state of a split or join step, respectively. `joinAndTrans` returns the empty stream `Nil` if the inner rectangle created by the join step does not satisfy the constraint C. `join` calls the partial function `joinV`, which raises the exception `Join` if it is applied to an undefined state, i.e. a state that does not satisfy the premise of one the axioms defining `join` (cf. 4.1). The exception is handled in the definition of `trans` by delivering the empty stream.

```
and join(state as (rect(0,0,l,h)::rect(x,0,l',h')::R,A,I))
      = if h' > h then (rect(0,0,l+l',h)::R,A,rect(x,h,l',h'-h)::I) else joinV(state)
|   join(state) = joinV(state)

and joinV(rect(0,0,l,h)::R,rect(0,y,l',h')::A,I)
      = if l' > l then (rect(0,0,l,h+h')::R,A,rect(l,y,l'-l,h')::I) else raise Join
|   joinV _ = raise Join
```

The two axioms for split (cf. 4.1) become definitions of two functions splitV and splitH, respectively:

```
and splitV(rect(0,0,l,h)::R,A,I)
       = let fun f(l') = (rect(0,0,l',h)::rect(l',0,l-l',h)::R,A,I)
         in if l < 2 then Nil else MapL(f)(interval(1,l-1)) end

and splitH(rect(0,0,l,h)::R,A,I)
       = let fun f(h') = (rect(0,0,l,h')::R,rect(0,h',l,h-h')::A,I)
          in if h < 2 then Nil else MapL(f)(interval(1,h-1)) end

and interval(x,y) = if x > y then nil else x::interval(x+1,y)
```

The main relation dissect transforms an initial state into admissable final states (cf. 4.1) and is thus transformed into a function that maps an initial state to the stream of all final states:

```
fun dissect(C,x,y,n) = let val r = rect(0,0,x,y)
                           fun f(l) = trans(C,n-1,l)([r],nil,nil)
                           fun sub(x,y) = if x < y then 0 else x-y
                       in (MapLS(f) o interval)(sub(n+1,x+y),n-2) end
```

The stream functions MapL and MapLS are defined in Section 1.2. In 3.1, areaDissect, brickDissect and propDissect generate streams of type $(int * int) list list$. Here the computed streams consist of states, which have the type $Rectlist * Rectlist * Rectlist$:

```
fun areaDissect(x,y,n) = let val a = (x*y) div n
                             val s = dissect(eqArea(a),x,y,n)
                         in (s,x+1,y+1) end
and eqArea(a)(rect(_,_,l,h)) = l*h = a

fun brickDissect(x,y,a) = let val n = ceiling(real(x*y)/real(a))
                              val s = dissect(leqArea(a),x,y,n)
                          in (s,x+1,y+1) end
and leqArea(a)(rect(_,_,l,h)) = l*h <= a

and propDissect(x,y,n,p) = let val s = dissect(proportioned(p),x,y,n)
                           in (s,x+1,y+1) end
and proportioned(p)(rect(_,_,l,h)) = p*l = h orelse p*h = l
```

Analogously to 3.1, areaDissect, brickDissect and propDissect return the size $(x+1, y+1)$ of the plane to be dissected in addition to the state stream s. The procedure next of 3.2 must be adapted to the fact that the above functions compute state streams instead of streams of point list partitions.

```
fun next(0) = finish()
|   next(n) = case !s_ptr of Nil => finish()
                           |  (R,A,I)&s => let ...
                                           in code:= !code^draw(R@A@I)^translate(l,h,x);
                                              ... end
```

5 Conclusion

This paper presented a case study of transforming logical design specifications with functions and relations into pure functional code. Since such specifications have a precise mathematical semantics, the specified algorithms can already be tested and verified on a rather abstract level (cf. Section 2.3). For compiling them into a pure logic language like Prolog, all functions must be flattened into relations (cf. Section 2.2). The translation into pure functional code works the other way around: relations used as multi-valued functions are compiled into single-valued functions, which enumerate all values pertaining to the same argument and store them into streams.

The case study deals with two configuration algorithms for generating rectangle dissections, which satisfy various constraints. The first algorithm constructs partitions of the point list representing the plane to be dissected. It branches out whenever a point is to be added to a recursively obtained partition of a sublist. Hence it works *bottom-up* insofar as dissections are built up point by point. This procedure ensures that each dissection is built up only once. The second algorithm starts out from a shape grammar, which defines transitions between states consisting of three lists of rectangles. The transitions decompose the plane to be dissected stepwise so that this algorithm works *top-down*. Since the underlying shape grammar is ambiguous, the same dissections may be built up several times.

Hence both algorithms follow quite different, almost complementary, design principles. The first is fast for dissections consisting of small rectangles, the second is more efficient if all dissections consist of a few big rectangles. Moreover, the translation into functional code raises different problems, which led us to general distinctions between *linear* and *nonlinear* nondeterminism and between *ambiguous* and *unambiguous* (axiomatic) function definitions (cf. Sections 3.1 and 4.2). While linear as well as nonlinear nondeterminism can be put rather schematically into functional code with the help of map functions on streams, the axioms of an ambiguous function definition have to be modified and rearranged in special ways in order to establish complete functional case analyses.

Future research should aim at general classifications of multi-valued function definitions and transformation rules, which admit a more schematic - maybe automatic - translation of logical specifications into stream-based functional programs. Configuration problems, which involve varying constraint parameters, make up a broad area of nontrivial applications for testing the power and the limits of such "logic-to-function" compilers.

Acknowledgement

I am grateful to Mihaly Lenart for drawing my attention to the field of configuration problems and shape grammars. Thanks to Michael Rexhäuser and Cornelius Rolf for developing and testing Prolog implementations of both dissection algorithms.

References

[BGM 88] P.G. Bosco, E. Giovannetti, C. Moiso, *Narrowing vs. SLD-Resolution*, Theoretical Computer Science 59 (1988) 3-23

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

[Car 93] C. Carlson, *Describing Spaces of Rectangular Dissections via Grammatical Programming*, Proc. 5th Int. Conf. CAAD Futures, North-Holland (1993) 143-158

[Gre 87] S. Gregory, *Parallel Logic Programming in PARLOG*, Addison-Wesley 1987

[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

[Pad 93] P. Padawitz, *Building Shelf Systems: A Case Study in Structured ML Programming*, Report No. 460, FB Informatik, Universität Dortmund 1993

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

[Pad 95] P. Padawitz, *Inductive Theorem Proving for Design Specifications*, Report No. 533/1994, FB Informatik, Universität Dortmund 1994, to appear in J. Symbolic Computation

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

[War 83] D.H.D. Warren, *An Abstract Prolog Instruction Set*, Technical Note 309, SRI International 1983