Describing Spaces of Rectangular Dissections via Grammatical Programming

Christopher Carlson

Research Institute for Symbolic Computation
Johannes Kepler University
4040 Linz
Austria

For the description of design spaces, grammatical programs (Carlson, 1993) offer several advantages over conventional grammars: (1) they integrate rewrite rules with more general nondeterministic functions, (2) they permit a range of programming styles from purely declarative to purely procedural, (3) they can describe spaces of constrained, parametric designs, and (4) they permit design space descriptions to be developed modularly. We demonstrate these features of grammatical programming by way of a simple example that generates spaces of rectangular dissections.

Keywords: generative systems, grammars, design spaces, functional logic programming.

1 Grammars and Grammatical Programming

Grammars have won favor as a formal method in design because they bridge the gap between the visual apprehension and expression of form preferred by the designer and the symbolic encoding required by the computer. Spatial grammars (Stiny, 1980, Carlson et al, 1991) are a "domain proximate" means of expression—they permit the designer to speak in very nearly the same form language as she uses in the design process. Grammars are also "process proximate" in that the way they generate designs resembles what designers do naturally: seek out form patterns and transform them or replace them with new ones. Thus grammars have the dual advantages of eloquence and comprehensibility.

But grammars suffer from several deficiencies that limit their usefulness in practice. While their rules may be parametric, the spaces they describe are limited to non-parametric designs. Grammars also provide no facilities for explicitly sequencing rules, a deficiency that leads to the construction of ad-hoc control mechanisms and obscure specifications. And grammars lack a means of encapsulating complexity—analogous to the subroutine in conventional programming languages—which is a prerequisite to developing and debugging design space descriptions of nontrivial complexity.

Grammatical programming (Carlson, 1993) seeks to address these deficiencies without sacrificing the intuitive character of grammars that has made them attractive in design. A grammatical program is a collection of mutually recursive definitions of nondeterministic functions. The functions may be viewed as being of roughly three types: (1) those that define a target algebra of designs and its subalgebras, (2) those that define primitive functions on designs, corresponding roughly to the individual rules of conventional grammars, and (3) those that define compound functions constructed from the primitives, corresponding roughly to the sets of rules in conventional grammars. A language of designs is obtained by applying a nondeterministic function to an initial design.

Unlike conventional grammars, grammatical programs give equal status to rewrite rules, compound rules ("grammars"), and arbitrary nondeterministic functions. Each type of function may be used as a building block in the construction of yet more complex functions of the other types. The resulting modularity facilitates the development of complex design space descriptions. Grammatical programs may be constructed, tested, and debugged piecewise, and may draw upon libraries of standard, debugged grammatical components.

The glue that binds grammatical components together and provides the fundamental control structures of grammatical programs are the operators of an algebra of nondeterministic functions: composition (<>), which sequences functions; union (<+>), which chooses nondeterministically between functions; iteration (<*>), which repeats a function a nondeterministic number of times; and failure (<~>), which tests whether a function yields any output for a given input. Using the algebraic operators to construct compound rules from primitive ones, one may program in styles ranging from purely declarative to purely procedural, according to the demands of the application.

Like conventional grammars, grammatical programs may include rewrite rules. In conventional grammars, rules are specified by their left- and right-hand sides, which give the conditions under which a rule applies to a target design and the effect on the target of applying the rule. To achieve full interchangeability of rules with arbitrary functions, grammatical programs take an "external" view of rules for which this internal structure is irrelevant. In grammatical programs, rewrite rules are obtained by taking "rewrite closures" of arbitrary functions. If a function maps a design D to a new design D', then its rewrite closure maps any design that contains D to a new design that has D replaced by D'. In effect, taking the rewrite closure of a function allows it to "seep inside" designs to act on their substructures. We characterize the applicability of rules to designs externally as well. In conventional grammars, a rule is said to apply to a target design if its left-hand side matches a substructure in the target design. In grammatical programs, a rule is said to apply to a target design if its action on the design yields any output.

The concept of rewrite closure is meaningful only in the context of a well-defined meaning of "substructure" of a design. Many structures of interest as representations of designs—sets, multisets, lists, trees, terms, and Stiny's shapes—have intuitive structure-substructure relationships. Functions on representations that lack meaningful concepts of substructure do not have meaningful rewrite closures; they may nevertheless serve as the target algebras of grammatical programs.

In the example of rectangular dissections we present in this paper, designs are represented as multisets of rectangles. These in turn are represented by algebraic terms over a vocabulary that includes variable symbols. The presence of variables in the vocabulary permits the representation of parametric designs. As in functional programming paradigms, grammatical program functions may be defined using argument patterns that restrict the domains of functions. Unlike in functional paradigms and conventional grammars, grammatical program functions may be applied to terms that contain variables, i.e., to parametric designs, and may yield parametric designs as well. Hence, grammatical programs can describe parametric design spaces.

We introduce grammatical programming in the following section by way of a simple example that generates rectangular dissections. The source code of the example is written in a prototype grammatical programming language called Grammatica (Carlson, 1993), whose syntax resembles that of Prolog. Grammatica is, in fact, a functional programming paradigm built on a constraint logic extension to Prolog. The constraint logic programming component (CLP(R), Jaffar et al, 1990) provides the unification mechanism that makes possible the generation of constrained, parametric design spaces. The functional programming component provides the machinery for algebraic specifications.

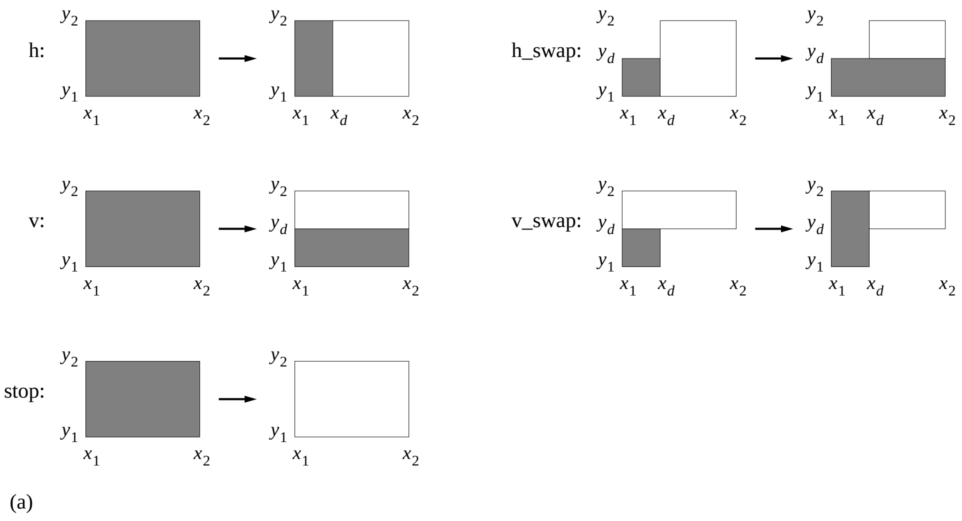
Nondeterministic functions return individual values chosen at random from sets of possible values. Operationally, Grammatica generates *every* value that a nondeterministic function may yield. Thus nondeterministic functions may be regarded as set-valued functions, and the sets as languages of designs.

2 The Dissection Problem

Figure 1(a) shows the rules of a conventional grammar that enumerates rectangular dissections. The grammar is adapted from a shape grammar by Flemming (1988); we discussed a similar grammar in an earlier paper (Carlson et al, 1991). Rules "h" and "v" partition rectangles horizontally and vertically; each application of "h" or "v" increases the number of rectangles in a dissection by one. Rules "h_swap" and "v_swap" rearrange rectangles to generate dissections with differing adjacency relationships. The shaded rectangle on the left-hand side of each rule constrains the rule to apply to the lower left rectangle of a developing dissection. This constraint maintains sufficient control over the dissection process to guarantee that each dissection is generated uniquely. The dissection process ends when the "stop" rule is applied to replace the shaded rectangle with a white one. Figure 1(b) shows the derivation of a typical member of the language of the grammar.

Our grammatical program is a cascade of two subprograms, the first of which works essentially like this conventional grammar. The output of the first subprogram is a language of dissections with constrained, but unfixed geometries. Each internal "wall" of a such a dissection is free to move within the bounds of the enclosing rectangle. The second subprogram in the cascade is chosen from among three schemes for fixing the dimensions of the component rectangles: one that assigns equal areas to rectangles, one that assigns equal proportions, and one that proportions each rectangle as if it were a brick or half-brick. The complete source code of the program is given in the Appendix.

Unlike many grammar paradigms, grammatical programming does not prescribe a



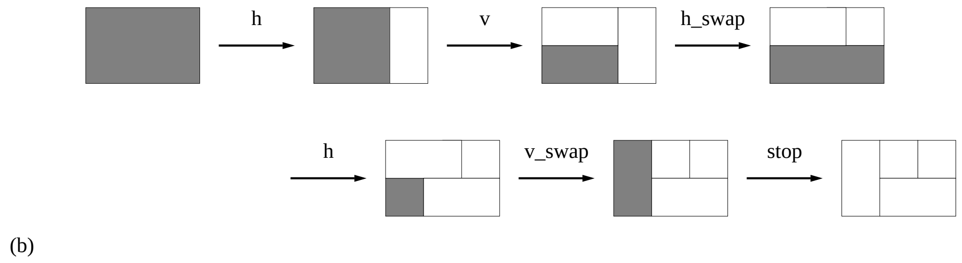


Figure 1. A conventional grammar that enumerates rectangular dissections: (a) rules, (b) typical derivation.

particular representation for designs. As in conventional programming languages, it is up to the author of a program to devise a representation suitable for the task at hand. We represent rectangular dissections as multisets (unordered lists) of rectangles. A typical dissection and its representation are shown in Figure 2. We use the Prolog bracket notation for lists to denote multisets. Each rectangle in a multiset is represented by an algebraic term of the form `rect(Label, X1, Y1, X2, Y2)` that gives the rectangle's label and the coordinates `(X1, Y1)` and `(X2, Y2)` of its lower left and upper right corners. A rectangle's label records its status in a dissection: the shaded rectangle is labeled `corner`, undimensioned rectangles `free`, and dimensioned rectangles `fixed`.

The program provides the `new_rect/5` function for creating new rectangles, defined by

```
min_dimension := 1.

new_rect(Label, X1, Y1, X2, Y2) :=
   (X2 - X1) >= min_dimension,
   (Y2 - Y1) >= min_dimension,
   rect(Label, X1, Y1, X2, Y2).
```

```
[
   rect(corner, 0, 0, 1, 1),
   rect(fixed,  1, 0, 3, 1),
   rect(fixed,  0, 1, 2, 2),
   rect(fixed,  2, 1, 3, 2)
]
```

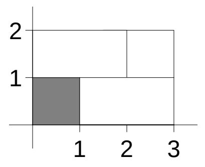


Figure 2. The symbolic representation of a dissection.

As this excerpt of code introduces many concepts in grammatical programming, we will go slowly in explaining it. Each unit of code of the form `head := body.`, called a program *statement*, gives a partial definition of the nondeterministic function named in the statement's head. The collection of statements whose heads share the same function name are collectively called that function's *definition*. We write `func/n` to denote the *n*-ary function `func`. Functions such as `min_dimension/0` that take no arguments are called *constants*.

A function *evaluates* to the quantities given in the bodies of the statements that comprise its definition. The constant `min_dimension/0`, for example, evaluates to `1`. Unlike conventional functions, nondeterministic functions may also evaluate to more than one value or none at all. A function that yields no values for some sets of arguments is said to be *undefined* at those arguments.

An expression involving `new_rect/5` is evaluated by first binding the variables `Label`, `X1`, `Y1`, `X2`, and `Y2` to the respective arguments in the function invocation. (Variables are captialized; other identifiers are lower-case.) Then the body of `new_rect/5`'s single definitional statement is evaluated. Expressions are always evaluated inside-out and from left to right. Comma-separated subexpressions of an expression are evaluated in turn, and the value of the last subexpression returned as the value of the expression as a whole. If any subexpression is undefined, then the expression as a whole is undefined. The body of `new_rect/5`'s definition begins with two constraints separated by commas. If both constraints are satisfied by the values bound to the variables, then the function returns the final `rect/5` term in the body of the definition. If either constraint is violated, the corresponding subexpression is undefined and hence the function as a whole is undefined, i.e., it returns no value. Thus `new_rect/5` is a partial function that returns a `rect/5` term given arguments that satisfy the dimensional constraints, and is otherwise undefined.

In general, expressions of the form

```
expr1, expr2, ..., exprN
```

may be regarded as equivalent to

```
if expr1 then
    if expr2 then
        ... then exprN
```

where "undefined" corresponds to "false" and "defined" to "true."

Functions (including constants) such as `rect/5` and `1` that are not explicitly defined but appear in the bodies of program statements are called *constructors*. Constructors represent the data that non-constructor functions operate upon. A constructor is for all practical purposes a function that evaluates to itself.

3 Enumerating Dissections

Returning to the dissection example, a space of dissections is generated by applying a nondeterministic enumeration function, which corresponds to a conventional grammar, to an initial dissection, which serves as the root of the design space. The enumeration function is constructed from primitive functions that correspond to the rewrite rules of a conventional grammar. The first step in constructing a rewrite rule is to define a function that maps a substructure, in this case a sub-dissection, to the structure that replaces it. In the case of the "h" rule, this function is `h_rect/1`, which maps a dissection containing one rectangle to a dissection containing two rectangles that partition the original rectangle horizontally:

```
h_rect([rect(corner, X1, Y1, X2, Y2)]) :=
   [new_rect(corner, X1, Y1, Xd, Y2),
    new_rect(free, Xd, Y1, X2, Y2)].
```

A rewrite rule, `h/1`, is obtained from `h_rect/1` by taking its rewrite closure using the built-in `rule/1` function:

```
h := rule(h_rect).
```

The effect of `h/1` on a target dissection is precisely that of a conventional rewrite rule: it yields the target dissection with one rectangle replaced by two rectangles that partition it horizontally. Note that the geometry of the replacement rectangles is not fixed. The `h_rect/1` function introduces the unbound variable `Xd` for the position of the partition, which is constrained by `new_rect/5` to lie between `X1` and `X2`.

The remaining rules `v/1`, `h_swap/1`, `v_swap/1`, and `stop/1` are similarly defined as rewrite closures of auxiliary functions.

Grammatical programming permits one to combine rewrite rules with other types of functions. *Filter* functions are particularly useful for exploring design spaces. A filter is a partial function that passes its argument unchanged if it meets specified criteria, and is undefined otherwise. The example program defines two filters that count rectangles. The first, `rect_count(N, Dissection)`, passes `Dissection` if it contains exactly `N` rectangles:

```
rect_count(N, Dissection) :=
   dissection_size(Dissection) = N,
   Dissection.
```

The second filter, `max_rect_count/2`, passes dissections that contain at most `N` rectangles.

Filters (and rewrite rules) are usually used in their *Curried* forms.[1] The idea is to let the application of an *n*-ary function to *m* arguments represent an $(n - m)$-ary function.

[1] After Haskell Curry, who made extensive use of the technique.

For example, application of the binary function `rect_count/2` to a single argument `N` yields the unary function `rect_count(N)` which passes dissections containing exactly `N` rectangles. Curried forms are applied to arguments using a postfix `//` notation that makes strings of rule or filter applications easy to read. The following expressions are equivalent in meaning, denoting the quantity `Dissection` when it contains 3 rectangles, and “undefined” otherwise:

```
rect_count(3, Dissection)
Dissection // rect_count(3)
Dissection // (3 // rect_count)
```

The rectangle-counting filters and previously defined rewrite rules are combined via the operators of a control algebra into one function that enumerates dissections containing `N` rectangles:

```
enumerate_dissections(N) :=
   ((h <+> v <+> h_swap <+> v_swap) <> max_rect_count(N))<*>
   <> rect_count(N)
   <> stop.
```

Informally, this definition functions as follows. The subxpression `(h <+> v <+> h_swap <+> v_swap) <> max_rect_count(N)` means “apply one of the `h`, `v`, `h_swap`, or `v_swap` rules, followed by the `max_rect_count(N)` filter.” The `<*>` operator repeats this operation zero or more times nondeterministically, yielding a dissection that contains at most `N` rectangles. Dissections that contain exactly `N` rectangles slip through the `rect_count(N)` filter to the `stop` rule, which completes the dissection process.

To enumerate a space of dissections, `enumerate_dissections/2` is applied to a root dissection. The example program provides `initial_dissection/2` to construct a root dissection comprising a single rectangle of specified dimensions centered at the origin:

```
initial_dissection(X, Y) :=
   [new_rect(corner, -X/2, -Y/2, X/2, Y/2)].
```

This function is combined with `enumerate_dissections/2` in a function that expresses the space of dissections of an `X`×`Y` rectangle into `N` component rectangles:

```
free_dissection_space(X, Y, N) :=
   initial_dissection(X, Y) // enumerate_dissections(N).
```

The space generated by `free_dissection_space(8, 8, 5)` is similar to the one depicted in Figure 3. It differs from the latter space in that the former contains parametric designs: the internal walls of dissections are constrained to lie within their containing rectangles, but are not further fixed. We describe the functions that fix the dimensions of dissections next.

4 Dimensioning Dissections

The `free_dissection_space/3` function describes spaces of parametric designs rooted at dissections given by `initial_dissection/2`. Dimensioning functions extend these spaces to spaces terminating in fully instantiated designs, i.e., in dissections so constrained that their geometries are uniquely determined. The spaces described by `free_dissection_space/3` serve, in effect, as the compound roots of spaces of dimensioned dissections.

We define dimensioning functions that give one-to-one, one-to-many, and many-to-one mappings from parametric dissections to dissections with fixed geometries. The first function, `dimension_by_area/2`, assigns equal areas to all of the rectangles in a dissection. Each free dissection is uniquely instantiated by `dimension_by_area/2`, thus the function serves as a canonical instantiation procedure. The second function, `dimension_by_proportion/2`, assigns to each rectangle in a dissection a specified proportion of length to width or width to length. Each free dissection has several distinct instantiations by `dimension_by_proportion/2`. The third dimensioning procedure, `dimension_as_bricks/2`, treats each rectangle in a dissection as a brick or half-brick, assigning to it a proportion of 1:2, 2:1, or 1:1. Some distinct free dissections map to the same instantiations under `dimension_as_bricks/2`.

The dimensioning functions repeatedly attach constraints to the rectangles in a dissection until every rectangle has been processed. The status of a rectangle is determined by its label: `free` rectangles need constraints, `fixed` rectangles already have them. Dimensioning rules use the `has_free_rects/1` filter to determine when all of the rectangles in a dissection have been constrained.

The `has_free_rects/1` filter is constructed like a rewrite rule by taking the rewrite closure of an auxiliary filter, `is_free_rect/1`, which passes `free`-labeled rectangles:

```
is_free_rect(rect(free, X1, Y1, X2, Y2)) :=
   rect(free, X1, Y1, X2, Y2).
```

Taking the rewrite closure of `is_free_rect/1` yields a filter that passes dissections containing `free` rectangles, and blocks those that do not:

```
has_free_rects := rule1(is_free_rect).
```

Like `rule/1`, `rule1/1` yields a rewrite rule from an arbitrary function. The resulting rule, however, rewrites only the first substructure it encounters in a target structure and ignores alternatives. The `rule1/1` closure is essential in situations such as rewriting each element in a set where a rewrite rule is to be applied at multiple loci in a target structure but where the result is not affected by the order in which the loci are rewritten. By using `rule1/1` instead of `rule/1`, the combinatorial explosion of duplicate designs that results from alternative but equivalent orderings of rule applications is avoided.

4.1 Dimensioning by Area

To dimension a dissection by area, the example program first constrains the area of every component rectangle to be equal to an (initially undetermined) value `A`. The resulting set of constraints is then solved by invoking a nonlinear constraint solver.

The `set_rect_area/2` function sets the area of one rectangle and returns the rectangle relabeled `fixed`:

```
set_rect_area(A, rect(free, X1, Y1, X2, Y2)) :=
   (X2-X1) * (Y2-Y1) = A,
   rect(fixed, X1, Y1, X2, Y2).
```

Taking the rewrite closure of `set_rect_area(A)/1` yields a rewrite rule that sets the area of a single rectangle in a dissection. The resulting rule is combined with the `has_free_rects/1` filter to construct a function that repeatedly constrains rectangles in a dissection until there are no free rectangles left, and then solves the resulting set of constraints:

```
dimension_by_area(A) :=
   rule1(set_rect_area(A))<*> <> has_free_rects<~> <> solve.
```

The failure operator (`<~>`) maps a unary function to a filter that passes those arguments at which the function is undefined. Recalling that `has_free_rects/1` passes dissections containing free rectangles and is otherwise undefined, the "failure" of `has_free_rects/1`, `has_free_rects<~>`, has the complimentary behavior: it passes dissections that do not contain free rectangles. Thus `dimension_by_area/2` repeatedly dimensions the free rectangles in a dissection until they have all have been relabeled `fixed`, at which point the dissection slips through the `has_free-_rects<~>` filter to the constraint solver invoked by `solve/1`.

Our Grammatica prototype solves linear constraints automatically, but puts nonlinear constraints on hold. The built-in `solve/1` function solves the constraints associated with an expression. When a dissection passes through the `has_free_rects/1` filter, its geometry is constrained to a unique configuration, which `solve/1` determines.

To generate spaces of equi-area dissections, `dimension_by_area/2` is applied to a root space of parametric dissections:

```
equi_area_space(X, Y, N) :=
   free_dissection_space(X, Y, N) // dimension_by_area(A).
```

The language generated by `equi_area_space(8, 8, 5)` is shown in Figure 3.

4.2 *Dimensioning by Proportion*

Dimensioning dissections by proportion is analogous to dimensioning them by area. Corresponding to `set_rect_area/2` and `dimension_by_area/2` are the functions `set_rect_proportion/2` and `dimension_by_proportion/2`:

```
set_rect_proportion(P, rect(free, X1, Y1, X2, Y2)) :=
   prop((X2-X1), (Y2-Y1), P),
   rect(fixed, X1, Y1, X2, Y2).

prop(X, Y, X/Y) := true.
prop(X, Y, Y/X) := true.

dimension_by_proportion(P) :=
   rule1(set_rect_proportion(P))<*> <> has_free_rects<~> <> solve.
```

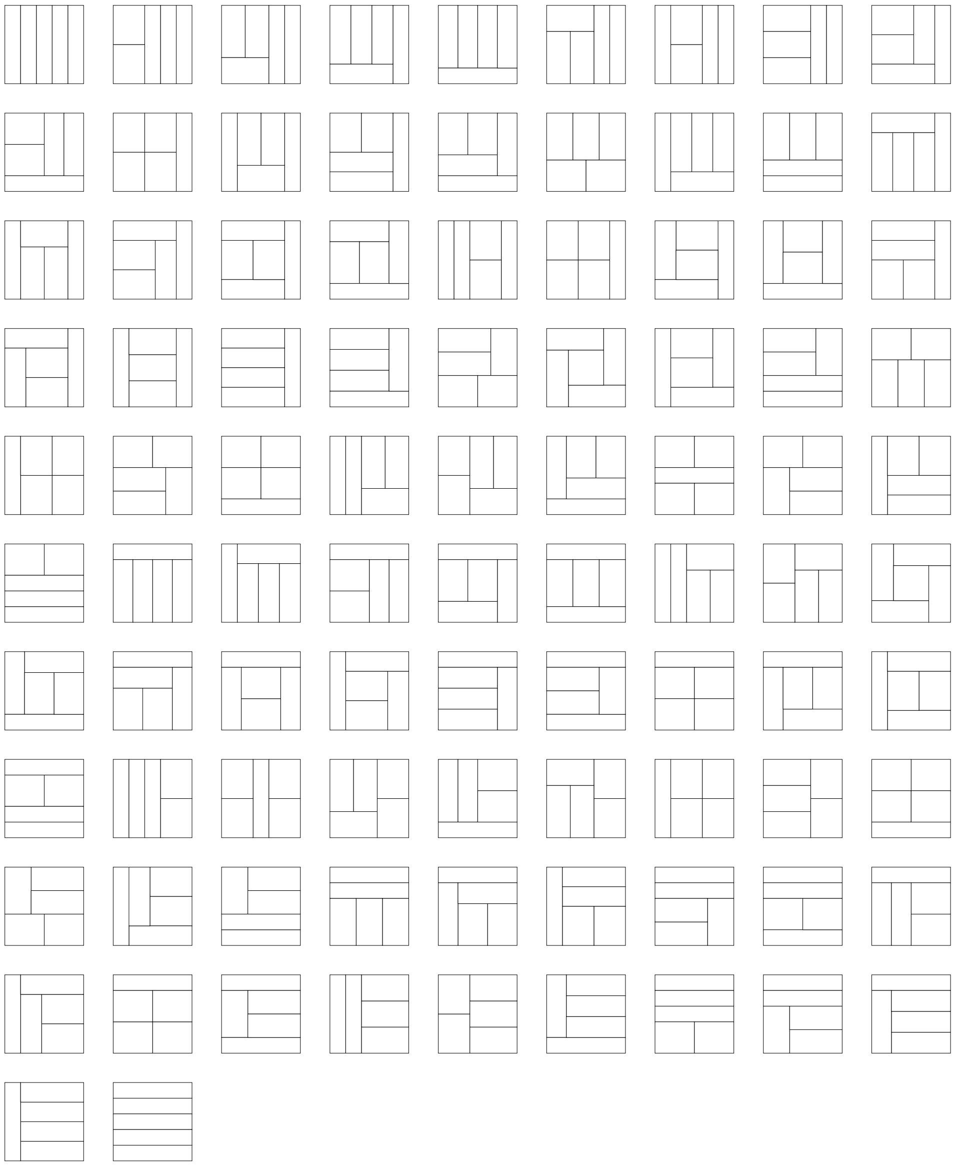


Figure 3. The 92 equal-area dissections of a square into 5 rectangles generated by `equi_area_space(8, 8, 5)`.

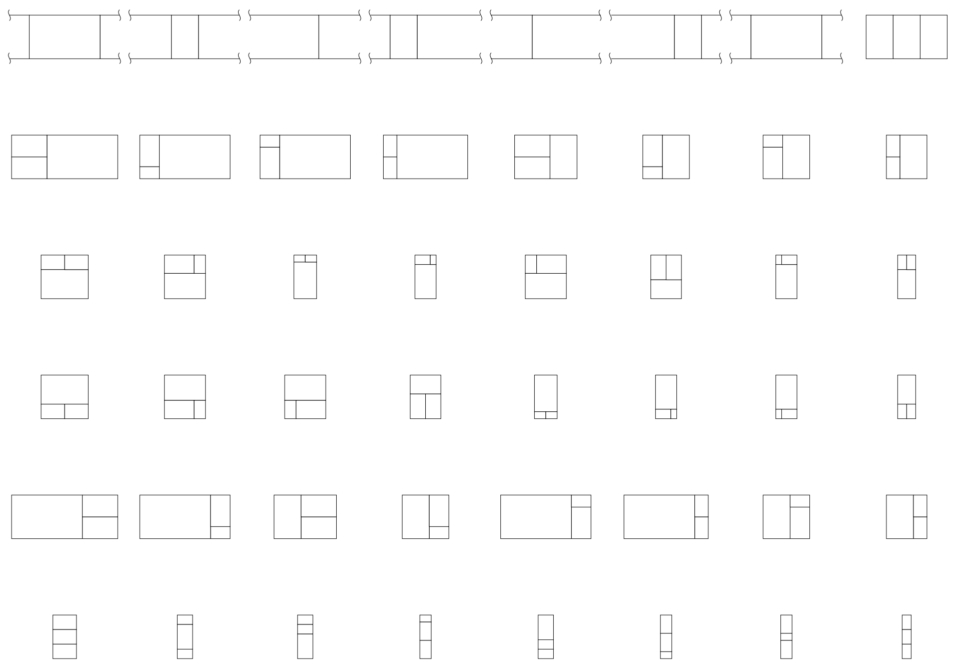


Figure 4. The 48 dissections of rectangles into golden rectangles generated by `proportioned_space(X, 8, 3, 1.6181)`. Each row contains the 8 dimensioned designs that derive from a single free design generated by `free_dissection_space(X, 8, 3)`. Dissections in the first row have been truncated on the left and right sides.

The auxiliary function `prop/3` is used to set the proportions of rectangles to one of two cases: length to width or width to length.

The `dimension_by_area/2` function is then combined with a root space of free dissections in the `proportioned_space/4` function, which generates `P`-proportioned, `N`-ary dissections of an `X` by `Y` rectangle:

```
proportioned_space(X, Y, N, P) :=
   free_dissection_space(X, Y, N) // dimension_by_proportion(P).
```

Figure 4 shows the language of dissections of rectangles into golden rectangles generated by `proportioned_space(X, 8, 3, 1.6181)`. The variable argument `X` leaves the horizontal dimension of the enclosing rectangle free to accommodate the packing of golden rectangles. Its value cannot be determined a priori, but is fixed by the proportioning of the component rectangles.

4.3 Dimensioning by the Proportions of Bricks

The third and last dimensioning function generates a family of dissections similar to the brick packings described by Carlson et al (1991), but by an entirely different mechanism. In the earlier example, a conventional grammar packed dimensioned bricks into a rectangle. In this example, the grammatical program assigns brick proportions to rectangles after they have been packed.

The function `make_rect_brick/2` is similar to `set_rect_proportion/2`:

```
make_rect_brick(U, rect(free, X1, Y1, X2, Y2)) :=
   brick_prop((X2-X1), (Y2-Y1), U),
   rect(fixed, X1, Y1, X2, Y2).

brick_prop(U, 2*U, U) := true.
brick_prop(2*U, U, U) := true.
brick_prop(U, U, U) := true.
```

The argument `U` is the unit length of a brick. Like `prop/3`, `brick_prop/3` is used to dimension a rectangle to one of three alternative proportions: `U`×`2U`, `2U`×`U`, or `U`×`U`.

`make_rect_brick/2` is incorporated in `dimension_as_bricks/2`, which dimensions all of the rectangles in a dissection and solves the resulting set of constraints:

```
dimension_as_bricks(U) :=
   rule1(make_rect_brick(U))<*> <> has_free_rects<~> <> solve.
```

This function is packaged with a root space of free dissections to yield the function `brick_space/2` which generates packings of 1×2 bricks in an `X`×`Y` rectangle.

```
brick_space(X, Y) :=
   ceil((X*Y)/2) => N,
   free_dissection_space(X, Y, N) <> dimension_as_bricks(1).
```

As each full brick has an area of 2 units, the expression `ceil((X*Y)/2)` gives the number of bricks in the rectangle, including one half-brick if `X`×`Y` is odd.

Figure 5 shows the packings of bricks generated by `brick_space(3, 3)`. The function generates duplicates of dissections such as that in the upper left corner of the figure in which four rectangles meet at a corner. Such dissections may be obtained as illustrated in Figure 6 from two distinct free dissections, one with a horizontally mobile wall, and one with a vertically mobile wall.

5 Assessment

In comparison to the grammar given by Carlson et al (1991), the function `brick_space/2` is less efficient and generates some duplicate packings. But we have constructed it easily from available components, and it is more flexible than the previous grammar. Exploring packings of 1×3 bricks, for example, requires only trivial, declarative modifications to the present program; the same adaptation of the previous, conventional grammar requires reworking its entire procedural structure. For “production” applications requiring efficiency, the previous grammar is clearly preferable. But for prototyping and exploring design spaces, the current example has clear advantages.

In general, modularity enhances exploration. The ability to construct new design space descriptions from readily available parts lowers the barriers between ideas and their realizations. Several features of grammatical programming improve the modularity of design space descriptions. The homogeneity of rewrite rules, filters, and general nondeterministic functions permits them to be freely mixed and interchanged. The control

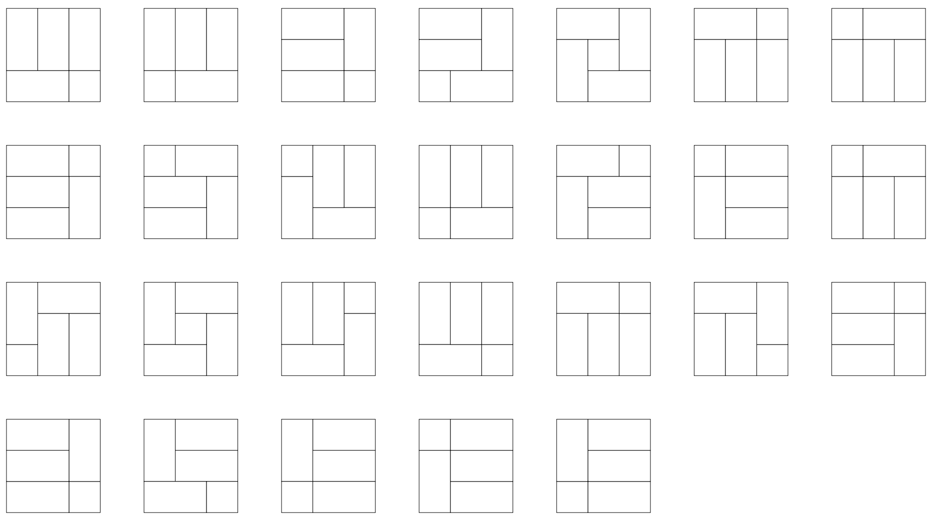


Figure 5. The 26 packings (including 8 duplicates) of bricks and half-bricks in a square generated by `brick_space(3, 3)`.

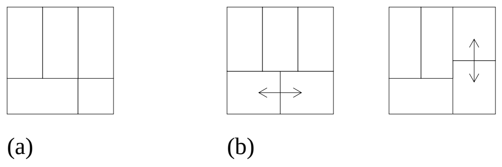


Figure 6. A brick packing (a) and two distinct free dissections (b) from which it derives.

algebra gives the programmer the ability to combine functions, and yet enough procedural control to avoid unwanted interactions among them. And the ability of grammatical programs to describe spaces of parametric designs contributes crucially to their modularity: this ability to defer geometric decisions permits the dissections program to be neatly divided into enumeration and dimensioning modules.

References

Carlson, Christopher, 1993. *Grammatical Programming: An Algebraic Approach to the Description of Design Spaces*. PhD thesis, Carnegie Mellon University.

Carlson, C., Woodbury, R. and McKelvey, R, 1991. “An introduction to structure and structure grammars.” *Environment and Planning B: Planning and Design* 18, p.417–426

Flemming, Ulrich, 1988. “Rules and representations in design.” Unpublished notes of a course given in Spring, 1988, in the Department of Architecture, Carnegie Mellon University, Pittsburgh, PA

Jaffar, J. et at, 1992. “The CLP(R) language and system.” *ACM Transactions on Programming Languages and Systems* 14, No.3, p.339–395

Stiny, George, 1980. “Introduction to shape and shape grammars.” *Environment and Planning B: Planning and Design* 7, p.343–351

Appendix: Source Code

```
%======================================================================
%  Enumeration Subprogram
%======================================================================
min_dimension := 1.

new_rect(Label, X1, Y1, X2, Y2) :=
   (X2 - X1) >= min_dimension,
   (Y2 - Y1) >= min_dimension,
   rect(Label, X1, Y1, X2, Y2).

%----------------------------------------------------------------------
%  Dissection Rules
%----------------------------------------------------------------------
h_rect([rect(corner, X1, Y1, X2, Y2)]) :=
      [new_rect(corner, X1, Y1, Xd, Y2), new_rect(free, Xd, Y1, X2, Y2)].
h := rule(h_rect).

v_rect([rect(corner, X1, Y1, X2, Y2)]) :=
      [new_rect(corner, X1, Y1, X2, Yd), new_rect(free, X1, Yd, X2, Y2)].
v := rule(v_rect).

h_swap_rect([rect(corner, X1, Y1, Xd, Yd), rect(free, Xd, Y1, X2, Y2)]) :=
      [new_rect(corner, X1, Y1, X2, Yd), new_rect(free, Xd, Yd, X2, Y2)].
h_swap := rule(h_swap_rect).

v_swap_rect([rect(corner, X1, Y1, Xd, Yd), rect(free, X1, Yd, X2, Y2)]) :=
      [new_rect(corner, X1, Y1, Xd, Y2), new_rect(free, Xd, Yd, X2, Y2)].
v_swap := rule(v_swap_rect).

stop_rect([rect(corner, X1, Y1, X2, Y2)]) :=
      [new_rect(free, X1, Y1, X2, Y2)].
stop := rule(stop_rect).

%----------------------------------------------------------------------
%  Filters
%----------------------------------------------------------------------
dissection_size([]) := 0.
dissection_size([X | Xs]) := 1 + dissection_size(Xs).
```

```
rect_count(N, Dissection) :=
   dissection_size(Dissection) = N,
   Dissection.

max_rect_count(Max, Dissection) :=
   dissection_size(Dissection) <= Max,
   Dissection.

%-----------------------------------------------------------------------
%  Enumeration Function
%-----------------------------------------------------------------------
enumerate_dissections(N) :=
   ((h <+> v <+> h_swap <+> v_swap) <> max_rect_count(N))<*>
   <> rect_count(N)
   <> stop.

%-----------------------------------------------------------------------
%  Initial Dissection
%-----------------------------------------------------------------------
initial_dissection(X, Y) :=
   [new_rect(corner, -X/2, -Y/2, X/2, Y/2)].

%-----------------------------------------------------------------------
%  Design Space
%-----------------------------------------------------------------------
free_dissection_space(X, Y, N) :=
   initial_dissection(X, Y) // enumerate_dissections(N).

%=======================================================================
%  Dimensioning Functions
%=======================================================================
%-----------------------------------------------------------------------
%  Filters
%-----------------------------------------------------------------------
is_free_rect(rect(free, X1, Y1, X2, Y2)) :=
   rect(free, X1, Y1, X2, Y2).

has_free_rects := rule1(is_free_rect).

%-----------------------------------------------------------------------
%  Dimensioning By Area
%-----------------------------------------------------------------------
set_rect_area(A, rect(free, X1, Y1, X2, Y2)) :=
   (X2-X1) * (Y2-Y1) = A,
   rect(fixed, X1, Y1, X2, Y2).

dimension_by_area(A) :=
   rule1(set_rect_area(A))<*> <> has_free_rects<~> <> solve.

equi_area_space(X, Y, N) :=
   free_dissection_space(X, Y, N) // dimension_by_area(A).
```

```
%----------------------------------------------------------------------
%  Dimensioning By Proportion
%----------------------------------------------------------------------
set_rect_proportion(P, rect(free, X1, Y1, X2, Y2)) :=
   prop((X2-X1), (Y2-Y1), P),
   rect(fixed, X1, Y1, X2, Y2).

prop(X, Y, X/Y) := true.
prop(X, Y, Y/X) := true.

dimension_by_proportion(P) :=
   rule1(set_rect_proportion(P))<*> <> has_free_rects<~> <> solve.

proportioned_space(X, Y, N, P) :=
   free_dissection_space(X, Y, N) // dimension_by_proportion(P).

%----------------------------------------------------------------------
%  Dimensioning By Brick Proportions
%----------------------------------------------------------------------
make_rect_brick(U, rect(free, X1, Y1, X2, Y2)) :=
   brick_prop((X2-X1), (Y2-Y1), U),
   rect(fixed, X1, Y1, X2, Y2).

brick_prop(U, 2*U, U) := true.
brick_prop(2*U, U, U) := true.
brick_prop(U, U, U) := true.

dimension_as_bricks(U) :=
   rule1(make_rect_brick(U))<*> <> has_free_rects<~> <> solve.

brick_space(X, Y) :=
   ceil((X*Y)/2) => N,
   free_dissection_space(X, Y, N) <>
   dimension_as_bricks(1).
```