Synthesis of Layout Engines from Relational Constraints

Thibaud Hottelier
UC Berkeley
tbh@cs.berkeley.edu

Rastislav Bodik
University of Washington
bodik@cs.washington.edu

Abstract

We present an algorithm for synthesizing efficient document layout engines from compact relational specifications. These specifications are compact in that a single specification can produce multiple engines, each for a distinct layout situation, *i.e.*, a different combination of known vs. unknown attributes.

Technically, our specifications are relational attribute grammars, while our engines are functional attribute grammars. By synthesizing functions from relational constraints, we obviate the need for constraint solving at runtime, because functional attribute grammars can be easily evaluated according to a fixed schedule, sidestepping the backtracking search performed by constraint solvers. Our experiments show that we can generate layout engines for non-trivial data visualizations, and that our synthesized engines are between 39- and 200-times faster than general-purpose constraint solvers.

Relational specifications of layout give rise to synthesis problems that have previously proved intractable. Our algorithm exploits the hierarchical, grammar-based structure of the specification, decomposing the specification into smaller subproblems, which can be tackled with off-the-shelf synthesis procedures. The new synthesis problem then becomes the composition of the functions thus generated into a correct attribute grammar, which might be recursive. We show how to solve this problem by efficient reduction to an SMT problem.

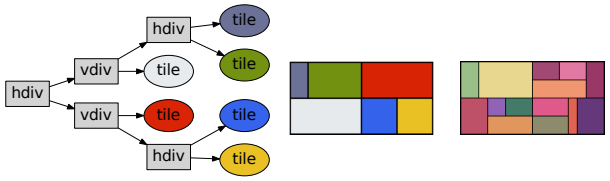
Categories and Subject Descriptors D.1.2 [*Programming Techniques*]: Automatic Programming—Program synthesis

General Terms Languages, Algorithms, Performance

Keywords Program Synthesis, Attribute Grammars, Layout

1. Introduction

Visual layout is the process of arranging visual elements such as paragraphs or images. A layout is computed by fixing the sizes and positions of some visual elements and using layout constraints to compute the sizes and positions of the remaining elements. The elements are organized in a *document tree*; the layout constraints are usually local and are attached to document nodes. For example, in Figure 1a, *hdiv* computes its width as the sum of its children's widths.



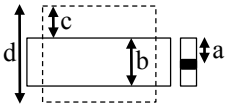
(a) Document (b) Layout of (a) (c) A larger treemap

Figure 1. A treemap visualization. A document tree (a), together with its layout (b). Figure (c) shows the layout of a larger and deeper document from the same layout language.

Modern layout languages (*e.g.*, CSS) are loosely based on *functional* attribute grammars (AG) [17] which define both syntactically legal documents and their layout semantics. Functional AGs are attractive because they can be solved efficiently using a *layout engine*, an evaluator with a fixed set of document tree traversals [22, 30]. However, the performance gained thanks to a fixed computation strategy comes at the cost of limited expressiveness. For instance, CSS always computes width before height. As a result, CSS 2.1 cannot express layouts such as treemaps (Figure 1) where *vdiv* derives width as a function of height[1].

We sidestep the limitations of (directional) functional constraints by specifying layout with (non-directional) relational constraints [15]. Relational constraints do not fix the computation strategy: the value "flows" between variables depending on which values happen to be known. As an example, consider a scroll box containing (i) a textbox (dashed rectangle), part of which is visible in a view port (solid rectangle); and (ii) and a scroll bar, which indicates the position of the textbox relative to the view port. The layout of the scroll box is specified with this relational constraint:

$$\frac{a}{b} = \frac{c}{d-b}$$



When the user moves the textbox, she fixes the textbox position c, necessitating recomputation of the scroll bar position given that we know the value of c. Conversely, when she moves the scroll bar, the textbox position is recomputed

[1] Treemaps are expressible in CSS using Flexboxes, an experimental extension gaining support in recent versions of web-browsers.

from the new value of a. Each such interaction triggers a different flow of computation, but maintains the same constraints.

However, efficient solving of constraint-based layouts remains a challenge. While the layout of a document can be computed with a general-purpose constraint solver, such solvers are up to 200-times slower than tree-traversal engines (see Section 7), which is insufficient for interactive settings, especially on slower, mobile devices.

We present a synthesis algorithm for translating relational layout specifications into efficient (*i.e.*, statically schedulable) functional attribute grammars. For simplicity, we illustrate synthesis on the flat constraint system of the scroll box. Our algorithm derives directional functional constraints for updating the document position ($c \leftarrow a(d-b)/b$) and for updating the scroll bar position ($a \leftarrow cb/(d-b)$). The key point is that both functional constraints can be derived from the single relational constraint. Relational constraints thus offer non-redundant encoding and freedom from a fixed computation strategy.

Single-Document Synthesis The most efficient layout engine is one created for a specific single document. The layout specification of a document is the conjunction of local relations from all tree nodes. The Comfusy algorithm [18] can translate such specifications to functions. This function would then solve the layout of the document. Scaling synthesis to large specifications is a challenge, however. So far, synthesis has been mostly limited to producing program fragments: Our experiments show that Comfusy scales to relations of up to 100 variables. However, layout specifications can include 10^4 program variables per document.

Modular Synthesis To scale synthesis to large specifications, we rely on the hierarchical structure of the specification, specifically a conjunction of smaller relations. In the case of a single document, each node carries a *block* of constraints. We decompose the synthesis problem at node boundaries, into smaller subproblems whose solutions (local functions) are composed to form a global function that implements the overall relation. We trade completeness for efficiency: modular synthesis cannot perform deduction across decomposition boundaries (only function composition), so we may fail to produce a global function when one exists. The reason is that smaller relations may not be functional, preventing us from producing the necessary local functions.

Grammar-Modular (GM) Synthesis So far we have outlined how modular synthesis can generate functions solving one particular relation, or equivalently one document. To be practically useful, we must synthesize a single layout engine generic enough to solve any document from a language of documents. We specify such languages using grammars of trees. Figures 1b and 1c show the layouts of two documents from a language of treemaps computed by the same synthesized engine. In essence, we generalize modular synthesis to accept not a fixed relation but a language of relational specifications represented as a *relational* attribute grammar (AG) [12]. The corresponding synthesized program is a *functional AG*. That is, the layout engine is a set of functions whose composition is syntax-directed by the document tree structure. We generalize modular synthesis to grammars of relations by handling alternative and recursive productions. To guarantee that our functional AGs are statically schedulable, we reject grammars with cyclic dependencies between attributes, thereby forbidding fixed-point computations.

Applications of GM Synthesis Even though GM synthesis was motivated by the challenges posed by layout, the techniques presented in this paper are generic: the algorithm is applicable to any hierarchical specification expressible as a relational AG. It could, perhaps, be used in modern hardware description languages (*e.g.*, Chisel [6]) where designs are trees of components with constraints on sizes, timings, etc.

We believe that small, domain-specific, *layout* languages (DSLL) can be both more expressive and more efficient than large—general-purpose—layout languages such as CSS. GM synthesis is part of the Programming by Manipulation (PBM) framework[2], a DSLL builder for non-programmers [14]. PBM infers layout specifications from user demonstrations, producing a relational AG which is then compiled to a tree-traversal engine using GM synthesis.

This paper makes the following contributions:

1. We present grammar-modular (GM) synthesis, a new technique enabling program synthesis to scale to previously intractable problems. We exploit the hierarchical structure of tree-shaped specifications to create a decomposition. We perform synthesis on each part individually, and combine the resulting functions into a program satisfying the overall specification.
2. We apply GM synthesis to generation of layout engines for DSLLs. We demonstrate empirically that our synthesized engines are sufficiently complete and are up to 200 times faster than Z3, a general-purpose constraint solver.
3. For constraints expressible as linear equations, we state a sufficient condition on the decomposition of the specification guaranteeing the completeness of GM synthesis.

2. Background and Motivation

In this section, we motivate our choice of non-directional constraints for layout specifications. We illustrate the flexibility and expressiveness of DSLLs built on relational AGs compared to CSS [8], perhaps the most widely used layout language based on functional AGs. In the process, we introduce the key layout concepts and describe the inputs and outputs of our GM synthesizer.

The Sidebar Layout Problem Assume we want to lay out a document composed of two panels: a sidebar and a main area.

[2] A demo of PBM is available at `pbm.cs.berkeley.edu`.

```
Inputs=hbox.α, hbox.w

S ::= hbox(B, M) with
  B.w ← hbox.α*hbox.w
  M.w ← (1 - hbox.α)*
        hbox.w
B ::= div() with
  div.h ← f(div.w) ★
M ::= <same as B>
```

(a) Overflowing f-AG (CSS)

```
S ::= hbox(B, M) with
  B.w + M.w == hbox.w
  B.h == hbox.h
B ::= bar() with
  bar.h == f(bar.w)
M ::= main() with
  main.h == f(main.w)
```

(b) Sidebar DSLL r-AG

```
Inputs=hbox.h, hbox.w

S ::= hbox(B, M) with
  M.w ← hbox.w - B.w
  B.h ← hbox.h
B ::= bar() with
  bar.w ← f⁻¹(bar.h)
M ::= main() with
  main.h ← f(main.w)
```

(c) Synthesized sidebar f-AG

```
Inputs=bar.w, main.w

S ::= hbox(B, M) with
  hbox.w ← B.w + M.w
  hbox.h ← B.h
B ::= bar() with
  bar.h ← f(bar.w)
M ::= main() with
  main.h ← f(main.w)
```

(d) Synth. resizing sidebar f-AG

Figure 2. Four layout languages for the sidebar problem. Capital letters are non-terminals, *hbox*, *div*, *bar*, and *main* are blocks. Language (a) is the essence of CSS. It always computes height as a function of width; thus it cannot express our sidebar design. Language (b) is non-directional which allows it to express our sidebar. From (b) and the input set $\{hbox.w, hbox.h\}$, we synthesize a functional AG (c) computing the sidebar layout. From (b), we can also produce an alternative functional AG recomputing the layout from a different set of inputs, for instance when the user resizes both panels (d).

Both contain malleable content (*e.g.*, images, text), whose aspect ratio can be altered. The corresponding document tree is made of a top-level horizontal divider with two children, one per panel. We want to make the sidebar wide enough to display all of its content on exactly one screen vertically without over/under-flow. The contents of the main panel, however, are allowed to overflow. To compute such a layout, one would first compute the width of the sidebar, given the screen height, and then compute the main area height, given the sidebar width.

Layout Specifications To specify the layout semantics of our document, we assign *blocks* to document nodes. Blocks define a set of attributes (*e.g.*, positions and sizes) which decorate document nodes. Blocks also place either update functions (functional AG) or non-directional constraints (relational AG) on these attributes. Some attributes can be marked as *input*; these are runtime values unknown at compile time, *e.g.* the size of an image, or the screen size. Solving the document layout amounts to computing the values of all attributes given input values, in accordance with the blocks' semantics.

Surprisingly, our simple example is impossible to implement with CSS. Figure 2a shows the simplified specification of the two CSS blocks relevant to our example. Since CSS is based on a functional AG, blocks semantics are expressed with functions. The sidebar must compute its width from its height. However, CSS always computes height as functions of width (★ in Figure 2a). The solving efficiency of CSS hinges on this restriction: it enables a tree-traversal engine where widths are computed in a traversal preceding all height computations. As such, even if CSS were extensible (it is not), it would be difficult to add a new sidebar block computing its width from its height[3].

By specifying the layout semantics with constraints instead of functions, we decouple the layout properties from their computation. The flow of computation can now be tailored to the particular type of layouts targeted by our DSLL. Figure 2b shows a relational AG defining two blocks (*bar*, *main*). A relationship between width and height is given, but which of these attributes is computed first is left open. Given a DSLL specified with a relational AG (Figure 2b) and a set of input attributes, our synthesizer outputs a functional AG (Figure 2c), capable of computing the layout of all derivable documents. Non-directional constraints gave us the flexibility to compute width from height for the sidebar and vice versa for the main panel. In practice, the relation between width and height (f in Figure 2) may not always be invertible, for instance if the content of the panel is a flow of text. This fact further motivates *domain-specific* layout languages over "one size fits all" languages: a tree layout (Figure 9b) should not be restricted by text-layout shortcomings.

In interactive layouts, the choice of input attributes depends on user actions. In the scroll box example, when the user moves the slide bar, its position becomes the input from which the layout engine recomputes the textbox position. From the same relational AG, we synthesize multiple layout engines, one per user interaction. Figure 2d shows another functional AG, also synthesized from Figure 2b, recomputing the layout when the user sets the width of both panels by dragging the middle separator. Relational AGs capture multiple layout scenarios in a single specification.

***Synthesis and* DSLLs** Adding new features to general-purpose layout languages is difficult because unexpected interactions between features take time to resolve. GM synthesis makes creating DSLLs quicker thus more practical. Our tool automatically synthesizes an efficient layout engine but also verifies the absence of ambiguities and inconsistencies: a layout engine is generated only when the specification is functional (i.e, deterministic) in the runtime inputs.

Paper Overview This paper is organized as follows: In Section 5, we introduce GM synthesis for single documents. Section 6 generalizes our algorithms to grammars of documents and discusses the completeness of our approach. Finally, we

[3] A CSS extension allowing such computations (Flexboxes) has been in the work for five years and is still not finalized: the specification is under review as "final working draft".

present our experimental results on synthesis of layout engines (Section 7) and discuss related work (Section 8).

3. Documents and Solvers

This section defines documents, the layout semantics, and the structure of solvers that we wish to synthesize.

Documents A document is a tree whose nodes are labeled with a terminal symbol. Figure 1a shows an example of a document tree with `hdiv`, `vdiv`, and `tile` nodes. The terminal symbol specifies the layout semantics of the node as described below.

Tree Grammars Document trees are generated by a regular tree grammar [9]. Tree grammars can be thought as generating derivation trees of context-free *word* grammars. Productions of tree grammars have the form `A ::= t(B,C)`, where `A`, `B`, `C` are non-terminals and `t` is a terminal. The tree in Figure 1a was generated from this grammar:

```
S ::= hdiv(T,T)      S ::= tile()
T ::= vdiv(S,S)      T ::= tile()
```

Attributes Each node is also labeled with a set of named attributes, such as `width` and `x`. The values of these attributes determine the semantics of a document. The values are given as the solution of a set of constraints. The layout of a document d is a valuation of d's attributes that satisfies all constraints in d.

Constraints Constraints are organized into *blocks*, which are pairs of attributes and the constraints imposed over them. We distinguish two kinds of blocks:

- A *node block* is a node-local constraint over attributes of a node. All nodes labeled with the same terminal share the same node block and thus have the same layout behavior. For example, all nodes that stack their children horizontally may be labeled with the terminal `hdiv` and their constraint forces horizontal layout.
- A *connection block* is a constraint between attributes of a parent node and the attributes of some its children. For example, the `height` of a node may be equal to `height` of its left child. We normalize our documents, so that connections are always equality constraints.

In the grammar below, productions are associated with constraints. The first constraint is a connection block $(a.x = T.x, \{a.x, T.x\})$, while the second is the node block $(x = f(y,z), \{x,y,z\})$ associated with all nodes labeled with the terminal `b`. We denote with b the block associated with the terminal `b`.

```
S ::= a(T) with a.x == T.x
T ::= b()  with b.x == f(b.y, b.z)
```

Connections and node blocks play equivalent semantic roles and, except in Section 6, where connections can create cycles, we do not need to distinguish between them. We will refer to them as blocks.

Structure of Constraints We assume that a constraint R is represented in CNF, i.e., $R = cl_0 \wedge \ldots \wedge cl_n$. This CNF structure will govern the decomposition of the synthesis problem, with the clauses cl_i becoming the smallest synthesis subproblems (see Section 5).

The techniques presented in this paper are independent of the logical theories used to express constraints. Our implementation used polynomial equalities and linear inequalities over reals, augmented with basic trigonometric functions and min/max operators (see Section 7). Empirically, we found such constraints expressive enough to specify a wide class of layouts and visualizations.

Functional Solvers We are interested in linear-time solvers which compute each attribute exactly once, using a statically computed function over other attributes. Such a solver performs no backtracking and we call it a *functional solver*.

In our algorithm, functional solvers are constructed from *oriented blocks* which are in turn constructed from the original blocks of the document. We now define blocks, oriented blocks, and functional solvers. Their use is illustrated in the subsequent section.

- *Blocks:* A *block* is a pair (R, V), where V is a set of attributes, and R is a relational constraint over V. Blocks model constraints at three levels of granularity: (i) documents, (ii) nodes, and (iii) sets of clauses that result from decomposition of larger blocks using their CNF structure. The constraints of the entire document d form the block $rel(d) = (R_d, V_d)$, where V_d are all attributes in d and R_d is the conjunction of all document constraints, *i.e.* all node blocks and all connection blocks. Blocks may share attributes, as shown in Figure 4, where blocks a and b share attributes x and y.
- *Oriented blocks:* An *oriented block* is a triple (R, V, I) where R are constraints over attributes V, as in a block. We distinguish the set $I \subseteq V$ of *input* attributes and the set of *output* attributes, which are the attributes in $V \setminus I$. Input attributes are special in that we know their values when solving the values of output attributes. In other words, the values of output attributes are solved after input attributes are set to particular constants.
- *Functional solvers:* A *functional solver* f is a quadruple (α, V, I, S), where V and I are attributes defined as in an oriented block, and α is an assignment $O := h_b(I)$ of values to output attributes; the values are computed from input attributes I. The set S maps the functional solver f to the set of CNF clauses satisfied with α. We say that the d-solver f *implements* an oriented block $b = (R, V, I)$ if the assignment α satisfies R.
- *d-Solver*: A *d-solver* for a document d is a functional solver $f = (\alpha, V_d, \{\}, S_d)$ that computes all attributes V_d of the document without requiring that any input values be provided, *i.e.* $I = \{\}$. The set of clauses satisfied by the assignment α must cover the document constraint $rel(d)$.

Functional Synthesis Functional synthesis [18, 24] is a procedure, denoted π that transforms an oriented block $b = (R, V, I)$ into a solver $f = (\alpha, V, I, S)$ that implements b. π returns the function h for the assignment $\alpha : O := h(I)$ if R is functional in the input attributes I. Otherwise, π fails. Functional synthesis can be implemented using existing techniques (see Section 7). Functional synthesis is the base-case for our modular synthesis, in that it translates the smallest oriented blocks into the corresponding solvers. Invoking π on small blocks addresses the problem that synthesizing a d-solver for a document d (*i.e.*, invoking $\pi(rel(d))$) is impractical for all but the most trivial documents, as shown in Section 7.

4. Overview of Modular Synthesis

In Section 3 we defined oriented blocks, as blocks with distinguished input attributes. The purpose of oriented blocks is to describe the value-flow structure of a document solver. The block for the document decomposes into smaller blocks. Attributes shared by blocks are outputs in one block and inputs in another. The solver proceeds by computing output attributes in a block b_i, which sets input attributes in the adjacent block b_j, which shares attributes with b_i. This allows solving output attributes in b_j, and so on. How does the process start? Some output attributes can be computed without setting any input attributes, typically those whose values are set by layout constraints, *e.g.* `width == 100`.

A functional solver has the same structure as a solver constructed from oriented blocks, in the sense that values flow from input to output variables. The difference lies in how output variables are computed: oriented blocks invoke a potentially backtracking solver while functional solvers compute the values using the function h_b.

Given a document d, we synthesize a d-solver in three steps (Figure 3):

1. decompose the specification ($rel(d)$) into smaller blocks along the conjuncts of the CNF formula; orient these blocks by selecting their input attributes;
2. perform synthesis locally, on each oriented block, obtaining *local* functional solvers; and
3. select and compose just enough solvers to construct a *global* solver computing all attributes of d, thus creating a d-solver.

4.1 Example of d-Solver Synthesis

Let us consider a document d comprised of two connected nodes labeled with blocks $a \stackrel{\text{def}}{=} (R_a, V_a)$ and $b \stackrel{\text{def}}{=} (R_b, V_b)$, with these specifications:

$$R_a \stackrel{\text{def}}{=} x = i \wedge i + z = y \wedge i = 0 \qquad V_a \stackrel{\text{def}}{=} \{x, y, z, i\}$$
$$R_b \stackrel{\text{def}}{=} x^2 = y \qquad V_b \stackrel{\text{def}}{=} \{x, y\}$$

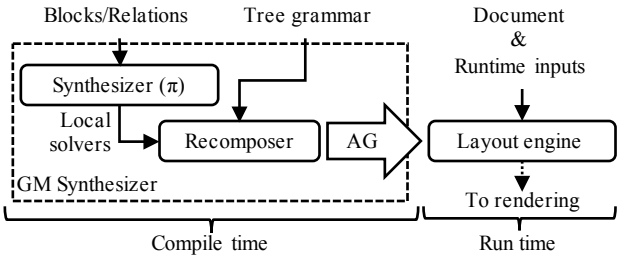


Figure 3. GM synthesizer. The first step of GM synthesis—decomposition—is not shown. The AG scheduler is out of the scope of this paper. Its output is the layout solver itself.

To simplify the example, we abstract away connections. Instead, the nodes share attributes x and y that would normally be joined with equality connection constraints.

The specification of the document, $rel(d)$, is simply $(R_a \wedge R_b, \{x, y, z, i\})$. To create a d-solver $(x, y, z, i := h(), \{x, y, z, i\}, \{\}, S)$, we must synthesize the function h that computes the attributes x, y, z, i.

Decomposition (Step 1) We perform two levels of decomposition. First, we treat each node block and each connection of $rel(d)$ separately, performing node-level decomposition. In our example, we end up with the two original blocks: a and b. Second, we decompose the blocks into smaller blocks, relying on the CNF structure of the constraints in the blocks. The result is a partition of the clauses in each CNF constraint. Good partitioning will reduce the size of partitions (grouping as few clauses as possible) while ensuring that the resulting constraints still encode a functional relation.

For our example document, block b cannot be decomposed further, while block a is decomposed into three blocks:

$$a' = (x = i, \{x, i\})$$
$$a'' = (i + z = y, \{i, y, z\})$$
$$a''' = (i = 0, \{i\})$$
$$b = (x^2 = y, \{x, y\})$$

Next, we orient the blocks by selecting attributes that act as their inputs:

$$a' = (x = i, \{x, i\}, \{i\})$$
$$a'' = (i + z = y, \{i, y, z\}, \{y, i\})$$
$$a''' = (i = 0, \{i\}, \{\})$$
$$b = (x^2 = y, \{x, y\}, \{x\})$$

To summarize, we (i) decomposed each block by partitioning its clauses; and (ii) selected input attributes for each block. These two local choices must be globally coordinated; in this example we assume that the choices are made by a cooperating oracle.

Local Synthesis (Step 2) Next we synthesize local functional solvers for blocks a', a'', a''', and b using the functional

synthesis procedure π.

$$\pi(a') \to f_1 = (x := i, \{x, i\}, \{i\}))$$
$$\pi(a'') \to f_2 = (z := y - i, \{z, y, i\}, \{y, i\})$$
$$\pi(a''') \to f_3 = (i := 0, \{i\}, \{\})$$
$$\pi(b) \to f_4 = (y := x^2, \{x, y\}, \{x\})$$

These local functional solvers implement their oriented blocks.

Recomposition (Step 3) The third step constructs a solver implementing the global specification $rel(d)$ by selecting a subset of such local solvers whose dependencies permit composition and collectively compute all attributes of the document, thus implementing $rel(d)$. This is the key step of the modular synthesis.

Since the oracle produced exactly the necessary solvers, we now merely need to order them to satisfy their dependencies. That is, for each local solver f, the attributes read must be computed by other solvers before f is applied. We encode solver dependencies using a hypergraph whose vertices are attributes and whose edges represent available local solvers (Figure 4). The source of each edge indicates the set of attributes read and its destination the set of attributes computed. A topological sort of the hypergraph reveals the order in which to compose local solvers. Here, by applying f_3 first, then f_1, then f_4, and finally f_2, we obtain the desired global functional solver.

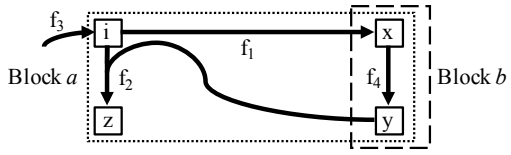


Figure 4. The hypergraph of dependencies among f_1, f_2, f_3. and f_4. Note that the local solver f_2 is represented by a hyperedge with two sources: i and y.

5. Modular Synthesis

Modular synthesis relies on the functional synthesizer π to perform synthesis on the subproblems created by decomposing $rel(d)$ and orienting the block resulting from the decomposition (Figure 3). The subsequent section generalizes the synthesis algorithm to accept a grammar of documents instead of a single fixed document.

Implementing the Oracle Let us take a step back to analyze the role of the oracle in Section 4.1. We relied on it twice during the local synthesis step: the first time to decompose the specification into small blocks, and the second time to orient blocks by selecting input attributes. Each of these local oracular decisions must be coordinated to achieve global properties not apparent at the local (*i.e.*, block) level:

- *Decomposition* When examining a block in isolation, we do not know how many local solvers are needed to compute all of its attributes. In our example, the attributes of block a are computed with two solvers in two steps: the value of y is required to compute z, but block b can compute y only if block a has already computed x. If we performed local synthesis directly on block a's relation (R_a), without decomposing it into the smaller blocks a', a'' and a''', we would be restricting ourselves to solving block a with a single local solver, which is not possible in our example.
- *Orientation* At the block level, we need to determine which attributes are known (inputs) and which attributes will be computed (outputs). The flow of computation is a property of the whole document and is unknown when synthesizing local solvers. In fact, the same node may be traversed multiple times by the global solver, each time invoking one local solver, like the node (labeled) a in our example. Intuitively, each subset of clauses corresponds to one "pass" of the global solver through the corresponding block.

We used the oracle to simplify our synthesis algorithm which *conceptually* relies on global reasoning to synthesize local solvers. In the implementation, we synthesize local solvers considering both all block decompositions and all block orientations. We then "implement" the oracle in the recomposition step, in which we must now select which local solvers to use. We perform the selection symbolically, by reasoning on a hypergraph summarizing all possible flows of computation. By selecting local solvers, we are indirectly making the same two decisions the oracle made: for each block, we partition its clauses into smaller blocks and choose input attributes.

5.1 The Algorithm

We formalize the three steps of GM-synthesis (decomposition, local synthesis, and recomposition) for a document d.

Decomposition (Step 1) We perform two level of decomposition. First, we decompose $rel(d)$ into blocks at nodes boundaries following the structure of the document, yielding a set of node blocks and connection blocks. This provides us with an initial decomposition where related constraints have been conveniently clustered together by the programmer. Then, we further decompose each block (R, V) by exploiting the CNF structure of R and exhaustively partitioning its clauses into smaller blocks (Algorithm 1).

There is no best granularity of decomposition: it is a trade-off between scalability and completeness of our approach. Finer decompositions lead to smaller relations and hence to more efficient local synthesis, but sometimes small relations are not functionalizable; they need to be conjoined with other relations to be functional. We discuss the completeness of GM synthesis in Section 6.3.

Local Synthesis (Step 2) To generate as many local solvers as possible, we enumerate all block orientations. That is, for each block (R, V) resulting from the decomposition step, we enumerate all input/output partitions of V. Then we apply the synthesis procedure π on each oriented block (R, V, I) to generate a local solver if R is functional in I (Algorithm 1).

Algorithm 1: Decompose a block and synthesize local functional solvers (steps 1 and 2).

Input: A block $b \stackrel{\text{def}}{=} (cl_0 \wedge \ldots \wedge cl_n, V)$
Output: A set of local solvers over attributes V

$L \leftarrow \varnothing$
foreach *subset* $S \subseteq \{cl_0, \ldots, cl_n\}$ **do**
  Let $V_S \subseteq V$ be the attributes appearing in S
  foreach *subset* $I \subseteq V_S$ **do**
    $b' \leftarrow (\bigwedge S, V_S, I)$
    if $\pi\,(b')$ *yields* (α, V_S, I, S) **then**
      Add (α, V_S, I, S) to L.
return L

Recomposition (Step 3) We reduce the problem of choosing and composing local solvers to finding a particular kind of spanning tree on a hypergraph. The hypergraph encodes a summary of all possible flows of computation between attributes of the document.

Definition 1 (Propagation Hypergraph)**.** *Given a document d and its block $rel(d) = (R_d, V_d)$, the propagation hypergraph of d is $H_d \stackrel{\text{def}}{=} (V_d, E)$ where V_d is the set of attributes of d and E is a set of local solvers. Each local solver (α, V, I, S) is represented with the hyperedge (I, O), where I is the set of source attributes and $O = V \setminus I$ the set of destination attributes.*

We construct the hypergraph H_d as shown in Algorithm 2: For each node n in d labeled with node block b, we instantiate the set of solvers synthesized for b on the attributes of n. We repeat the same process for each connection block c in d.

Algorithm 2: Construct a propagation hypergraph encoding all possible compositions of local solvers.

Input: A document d with $rel(d) = (R_d, V_d)$
Output: A propagation hypergraph (V_d, E) of d

$E \leftarrow \varnothing$
foreach *node n in d labeled with node block b* **do**
  Add $\{(I, V \setminus I) \mid (\alpha, V, I, S) \in \text{Algo1}(b)\}$ to E.
foreach *connection block c in d* **do**
  Add $\{(I, V \setminus I) \mid (\alpha, V, I, S) \in \text{Algo1}(c)\}$ to E.
return (V_d, E)

Before we define the d-solver in terms of paths in H_d, let us note the following two facts about the propagation hypergraph H_d. First, each hyperpath [5] encodes a solver computing its destination attributes using its source attributes as inputs.

Lemma 1. *Each acyclic hyperpath $p = f_0 \ldots f_n$ in H_d encodes a solver $f_p \stackrel{\text{def}}{=} (\alpha_p, V_p, I_p, S_p) = f_0 \circ \ldots \circ f_n$. Let V_i, I_i be respectively the sets all attributes and input attributes of f_i, the ith solvers in p. Let $O_i = V_i \setminus I_i$ and $O_p = \bigcup_{0 \leq i \leq n} O_i$. We have:*

1. $V_p = \bigcup_{0 \leq i \leq n} V_i$
2. $I_p = (\bigcup_{0 \leq i \leq n} I_i) \setminus O_p$
3. *α_p assigns attributes O_p from inputs I_p.*

We can conclude directly from the definition of hyperpaths [5] that:

1. The dependencies of each local solver on the path are satisfied. For each solver f_i with $i > 0$, we have $I_i \subseteq \bigcup_{0 \leq j \leq i-1} O_j \cup I_p$.
2. Each attribute is computed at most once: For any pair of solvers f_i and f_j in p such that $i \neq j$, we have $O_i \cap O_j = \varnothing$.

Lemma 2. *Let $p = f_0 \ldots f_n$ be an acyclic hyperpath in H_d representing f_p. Let $(\alpha_i, V_i, I_i, S_i)$ be the ith solver in p. Then f_p implements $S_p = \bigwedge_{0 \leq i \leq n} S_i$. We say that f_p implements* all clauses traversed.

Lemma 2 follows directly from the fact that, by construction, each local solver $(\alpha_i, V_i, I_i, S_i)$ implements S_i. Finally, let us define the subset of paths which can be executed.

Definition 2 (Executable Path)**.** *A hyperpath p in H_d is* executable *iff its solver has no external inputs. That is, the solver $f_p = (\alpha_p, V_p, I_p, S_p)$ is such that $I_p = \varnothing$.*

We are now ready to state under which conditions a hyperpath encodes a d-solver. That is, a global solver which implements $rel(d)$.

Definition 3 (Executable Covering Spanning Tree)**.** *The hyperpath p is an* executable covering spanning tree *iff all of the following three conditions hold: (i) p is executable; (ii) p is a spanning tree; and (iii) p traverses all clauses of $rel(d)$. We call the third condition* coverage.

Theorem 1. *Each executable covering spanning tree p in H_d encodes a global solver which implements $rel(d)$.*

Since p is an executable spanning tree, it follows that both $I_p = \varnothing$ and $O_p = V_p = V_d$. From the coverage condition and using Lemma 2, we conclude that f_p computes all attributes of d.

The existence of a executable covering spanning tree p in H_d implies that the valuation of attributes computed by p is unique. That is, the layout of d is deterministic.

Theorem 2. *If there exists an executable covering spanning tree in H_d, then R_d is functional (with respect to an empty input set).*

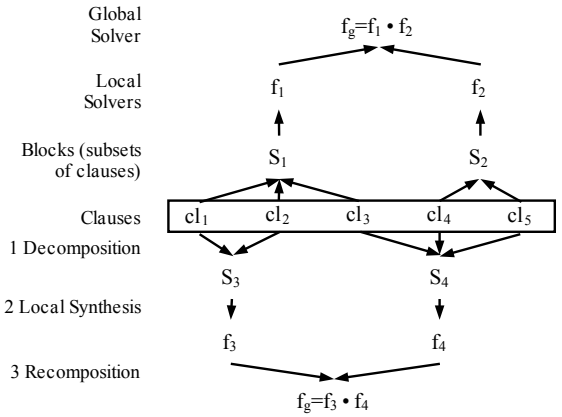


Figure 5. The three steps of GM synthesis. This diagram shows that two distinct decompositions can lead to syntactically different, yet semantically equivalent, d-solvers.

Since every local solver composing the covering spanning tree stems from a functional relation (with respect to block-level inputs), one can show that the set of all traversed clauses is functional too. Note that there may exist multiple covering spanning trees. Each such tree encodes a semantically equivalent global solver, but they may differ syntactically (Figure 5).

The converse is not true; GM synthesis assumes that the global solver is expressible as compositions of local solvers.

Together, Theorems 1 and 2 show that our approach is correct: the d-solvers synthesized always fulfill the specification. Note that finding a spanning tree in a hypergraph is NP-complete [32]. In the next section, we explain how to encode the search for a d-solver using SMT [7] after generalizing our approach to languages of documents.

6. Grammar-Modular Synthesis

We now generalize the modular synthesis technique to grammar-modular synthesis. Our goal is to translate a grammar of documents $\mathcal{G}$ into an $\mathcal{L}$-solver that accepts any document $d \in L(\mathcal{G})$. Technically, the $\mathcal{L}$-solver is a Knuth-style AG (whose rules are functions) [17]. In contrast, the document grammar $\mathcal{G}$ is a relational AGs [12], whose rules are relational constraints (provided in blocks). We start by formally defining document grammars and language solvers.

Definition 4 (Document Grammar)**.** *A* document grammar *$\mathcal{G}$ is a regular tree AG annotated with node and connection blocks constraining its attributes. A* language of documents *$\mathcal{L}$ is the set of documents generated by a document grammar.*

Definition 5 ($\mathcal{L}$-Solver)**.** *Let $\mathcal{L}$ be the language of documents induced by the document grammar $\mathcal{G}$. An $\mathcal{L}$-solver is a functional AG $\mathcal{G}'$ that is equivalent to $\mathcal{G}$, and there exists a static evaluation schedule s for $\mathcal{G}'$. The grammar $\mathcal{G}'$ and the schedule s define a d-solver for every document $d \in \mathcal{L}$.*

Two AGs are equivalent iff they generate the same set of trees and compute the same attribute values for each tree.

Example Consider the following document grammar with three blocks $a \stackrel{\text{def}}{=} b_1 \stackrel{\text{def}}{=} (true, \{x\})$, and $b_2 \stackrel{\text{def}}{=} (x = 0, \{x\})$.

```
S ::= a(T)  with  a.x == T.x
T ::= b1(S) with b1.x == S.x + 1
T ::= b2()  with b2.x == 0
```

Listing 1. A simple document grammar

The equivalent functional AG can be obtained simply by turning equality constraints into assignments. Both grammars are recursive and thus generate an unbounded number of documents. But all such documents can be solved with the same (*i.e.*, static) evaluation schedule: a single bottom-up pass.

To construct a $\mathcal{L}$-solver from $\mathcal{G}$, we compute: (i) a sufficient subset of local solvers and (ii) a total order over attributes. The total order prevents cyclic dependencies, which guarantees that the resulting functional AG is statically schedulable. The key differences with Section 5 are: (i) we now operate on a language of documents instead of a single fixed document; and (ii) we present an efficient way to construct $\mathcal{L}$-solvers using a reduction to SMT [7].

6.1 Overview of $\mathcal{L}$-Solvers Synthesis

Our generalization of modular synthesis is based on the observation that a grammar is a document in that it has the same structure except that (i) non-terminals may have alternative productions, selecting among possible children subtrees; and (ii) some productions introduce recursion, causing non-terminals to appear in the document multiple times. The synthesis for grammars will thus need to produce solvers that work no matter what productions have been chosen for a document, and no matter how large the tree is due to recursion.

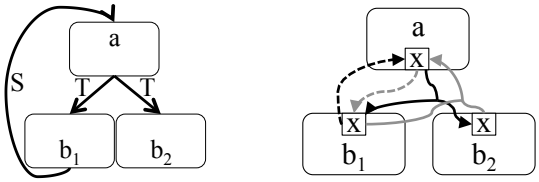
We synthesize an $\mathcal{L}$-solver in three steps:

Create Grammar Graph (Figure 6a) We create a graph representation of the grammar. The nodes are terminals and edges are productions that refer to those terminals. Each production is represented at least once, so that the summary graph contains all constraints of the grammar.

Some nodes (terminals) are replicated to offer more scheduling freedom among the productions of a non-terminal. In particular, the attributes in one production will be able to have different flows of computation than attributes in the other.

The graph is produced by systematically unrolling the grammar until every production is taken at least once. Recursive grammars produce a cyclic graph; some productions (*e.g.*, production S in Figure 6a) will create back-edges.

Note that a grammar may have multiple graph representations. In practice, choosing the grammar graph did not affect the performance or completeness of our algorithm.



(a) The grammar graph (b) The propagation hypergraph

Figure 6. The grammar (a) and propagation (b) graphs for Listing 1. Dashed arrows represent back-edge connections; the plain arrows are normal connections. We synthesize an $\mathcal{L}$-solver by selecting the two gray connections. The resulting cycle is due to our encoding of recursion in the grammar graph and does not represent cyclic attribute dependencies.

Create Propagation Graph (Figure 6b) We lift the grammar graph into the propagation hypergraph H_G analogous to Definition 1 in Section 5.1, where nodes are attributes and hyperedges are local solvers. The procedure simply invokes Algorithm 2 which was designed to work correctly not only for documents but also for cyclic grammar graphs. We view the grammar graph as a meta-document; the propagation hypergraph that we obtain with Algorithm 2 will create correct hyperedges for multiple productions of a non-terminal as well recursion.

For non-terminals with multiple productions, we must ensure that, for all productions, values flow in the same direction (either up or down) through each connection. In Listing 1, block a may have either block b_1 or b_2 as its child. Attribute $a.x$ is connected to either $b_1.x$ or $b_2.x$. Algorithm 2 encodes the connection block `a.x==T.x` with two hyperedges (one with two destinations and one with two sources), forcing values to flow either up or down through both derivations (Figure 6b).

Select Local Solvers Subject to Static Schedulability As in Section 5, we select a subset of local solvers from the propagation graph. To handle grammars producing more than one document, we modify our definition of executable covering spanning trees (Definition 3) to allow some cycles.

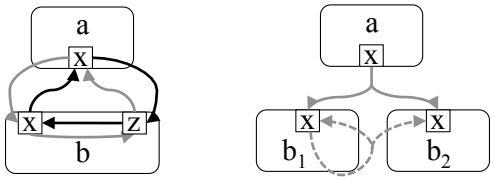
We distinguish two types of cycles in the propagation graph: (i) cycles representing recursion in the grammar; and (ii) dependence cycles among attributes. Recursion cycles (*e.g.*, gray cycles in Figures 6b and 7b) are the direct consequence of our decision to represent recursion with back-edges in the grammar graph. They unwind in all documents generated by the grammar. As such, they are not cycles in documents. Dependence cycles represent infeasible execution flows or fixpoint computations. To ensure static schedulability of the resulting grammar, we compute a total order over attributes while ignoring back-edges.

We illustrate dependence cycles on the following grammar with two blocks $a \stackrel{\text{def}}{=} (\{x\}, true)$ and $b \stackrel{\text{def}}{=} (\{x, z\}, x = z)$:

```
S ::= a(T)   with a.x == T.x ∧ a.x == T.z
T ::= b()    with b.x == b.z
```

Listing 2. An unschedulable grammar

Figure 7a shows a spanning (Definition 3) but unschedulable functional AG. All cycles in this grammar are dependence cycles. In fact, this grammar does not admit an $\mathcal{L}$-solver.



(a) Unschedulable AG (b) Schedulable AG

Figure 7. The propagation hypergraph of Listing 2 and a selection of local solvers (gray edges) that is not a schedulable AG (a). Both cycles (gray & black) are dependence cycles. A selection of local solvers for Listing 3 that is schedulable (b). Dashed arrows represent back-edge connections. Both the cycle around $b_1.x$ and the double-assignments on $b_1.x$ and $b_2.x$ unwind in all documents.

To satisfy all constraints, each attribute must be assigned exactly once. In Definition 3, the spanning tree guarantees single-assignment. This follows directly from properties of spanning trees in hypergraphs [32]. To allow recursion cycles, we need to modify the spanning property to allow some attributes to be assigned multiple times in the propagation hypergraph.

Recall that the propagation hypergraph is a finite summary of all derivable documents. As a result, an attribute may appear to be assigned twice in the propagation hypergraph: once "directly" by a normal connection, and once "inductively" by a back-edge connection. However, each document node will have exactly one parent: either the back-edge parent or the normal parent. As such, both connections must agree on the mode.

Consider the following grammar blocks $a \stackrel{\text{def}}{=} (\{x\}, x = 0)$ and $b_1 \stackrel{\text{def}}{=} b_2 \stackrel{\text{def}}{=} (\{x\}, true)$:

```
S ::= a(T)   with  a.x == T.x - 1
T ::= b1(T) with b1.x == T.x - 1
T ::= b2()   with b2.x == T.x - 1
```

Listing 3. A schedulable grammar (one top-down pass)

Each terminal computes its height in the document tree. Figure 7b shows the $\mathcal{L}$-solver solving it with a single top-down pass. On all documents generated by the grammar, the recursion cycles unwind; thus no document attributes are assigned twice.

6.2 Synthesis using SMT

Finding a spanning tree in a hypergraph (Theorem 1) is NP-complete [32]. In this subsection, we show how we leverage SMT [7] solvers to synthesize $\mathcal{L}$-solvers efficiently. In essence, the SMT solver is the oracle used in Section 4.1.

From the propagation graph, we construct an SMT formula ϕ whose models encode both attribute modes and a subset of local solvers. Together, they define a statically schedulable functional AG, *i.e.*, an $\mathcal{L}$-solver. We break our encoding in four parts: (i) coverage; (ii) modes; (iii) schedulability; and (iv) the spanning property. We explain them individually.

$$\phi := \phi_{\text{Cov}} \ \wedge \ \phi_{\text{Modes}} \ \wedge \ \phi_{\text{Sched}} \ \wedge \ \bigwedge_{x \in V} \phi_{\text{Span}}(x)$$

Coverage The coverage property (Definition 3) remains essentially the same when generalizing to language of documents. Intuitively, all clauses must be traversed by a local solver to ensure that all constraints are satisfied.

Given a language of document $\mathcal{L}$ induced by the grammar $\mathcal{G}$, let $d_{\mathcal{G}}$ be its grammar graph (cyclic meta-document). By construction, $d_{\mathcal{G}}$ contains all blocks, thus all constraints of $\mathcal{G}$. As a result, $rel(d_{\mathcal{G}})$ encodes the specification of all documents in $\mathcal{L}$.

Let $rel(d_{\mathcal{G}})$ be the block (R, V) and let $H_{\mathcal{G}} = (V, E)$ be the propagation hypergraph of $d_{\mathcal{G}}$. Recall that V is the set of all attributes of $\mathcal{G}$, and E the set of all local solvers, resulting from applying local synthesis on either node blocks or connection blocks. Each local solver $(\alpha, V, I, S)_f \in E$ is encoded with one boolean flag e_f, which is true iff f is used in the $\mathcal{L}$-solver; we say that f is *selected*.

To guarantee that the $\mathcal{L}$-solver implements $rel(d_{\mathcal{G}})$, the local solvers selected must traverse all the clauses of R.

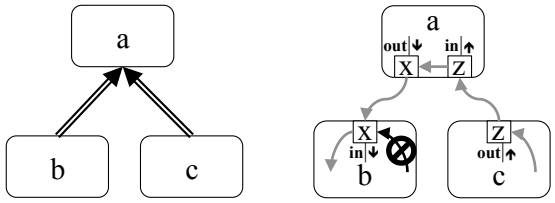
$$\phi_{\text{Cov}} \stackrel{\text{def}}{=} \bigwedge_{cl \in R} \ \bigvee_{e_f \in \theta(cl)} e_f$$

$$\theta(cl) \stackrel{\text{def}}{=} \{e_f \mid (\alpha, V, I, S) \in E \wedge cl \in S\}$$

Modes To ensure that $\mathcal{L}$-solvers are valid compositions of local solvers, the value of each attribute must either be computed inside its node block or passed by a connection block, but not both (Figure 8). We compute the *mode* of every attribute and every connection to enforce this invariant. In functional AGs, the mode represents how a value is propagated [17]: either *inherited* or *synthesized*. In Listing 1, attributes $a.x$, $b_1.x$, $b_2.x$ and the connection `a.x==T.x` are all synthesized.

Let $F \subseteq E$ be the subset of local solvers derived from node blocks. Let C be the set of connection blocks of $d_{\mathcal{G}}$. Recall that connection blocks are always equality constraints. For simplicity, we denote by $(A, B)_c$ the connection block $c \in C$ imposing an equality constraint on the set of parent attributes A and the set of children attributes B. For example, in Listing 1, the connection block `a.x==T.x` becomes $(\{a.x\}, \{b_1.x, b_2.x\})$.

We encode the mode of each attribute $x \in V$ with one boolean m_x and we encode the mode of each connection $(A, B)_c \in C$ with one boolean m_c. Both booleans are either inherited ($\downarrow$) or synthesized ($\uparrow$).



(a) A document (b) Attribute modes

Figure 8. Grammar attributes modes and their dual block modes. Consider a grammar producing a single document constituted of a block a with two siblings b and c (a). We show the modes imposed by selecting the gray local solvers (b). If $b.x$ is computed by its parent (inherited), the local solver computing $b.x$ in b (crossed out arrow) cannot be selected.

For attributes, it is sometimes more intuitive to reason with "block" modes (in or out), a dual of grammar modes ($\downarrow$ or $\uparrow$). Block modes are analogous to modes of logic programs: attributes marked in are computed outside the block and propagated to it by connections; attributes marked out are computed within the block by a local solver. Figure 8 shows both the grammar and block modes for a simple document.

We define $bm(x)$, the function converting the grammar mode of attribute x to its block mode.

$$bm(x) \stackrel{\text{def}}{=} \begin{cases} in & \text{if } \exists (A, B)_c \in C.\ (x \in A \wedge m_x = \uparrow) \ \vee \\ & \qquad\qquad (x \in B \wedge m_x = \downarrow), \\ out & \text{otherwise.} \end{cases}$$

We can now state the relationship between connection modes, attribute modes, and selected local solvers. Since connections are equality constraints, we synthesize exactly two local solvers per connection, one for the inherited direction, and one for the synthesized direction. Let $\gamma_\downarrow$, $\gamma_\uparrow$ be two functions mapping connections to their inherited/synthesized local solver, respectively.

$$\phi_{\text{Modes}} \stackrel{\text{def}}{=} \bigwedge_{(A,B)_c \in C} \phi_{\text{Conn}}((A, B)_c) \ \wedge \ \bigwedge_{(\alpha,V,I,S)_f \in F} \phi_{\text{Fun}}((\alpha, V, I, S)_f)$$

$$\phi_{\text{Conn}}((A, B)_c) \stackrel{\text{def}}{=} \big(m_c = \downarrow \Rightarrow e_{\gamma_\downarrow(c)} \wedge \textstyle\bigwedge_{x \in A \cup B} m_x = \downarrow\big) \ \wedge \\ \big(m_c = \uparrow \Rightarrow e_{\gamma_\uparrow(c)} \wedge \textstyle\bigwedge_{x \in A \cup B} m_x = \uparrow\big)$$

$$\phi_{\text{Fun}}((\alpha, V, I, S)_f) \stackrel{\text{def}}{=} e_f \Rightarrow \bigwedge_{x \in V \setminus I} bm(x) = out$$

Notice that we only constrain the output of local solvers from node blocks. Constraining their inputs would prevent chaining of local solvers within the same block, preventing the $\mathcal{L}$-solver from invoking multiple local solvers during the same traversal.

Schedulability To guarantee that the the resulting $\mathcal{L}$-solver is statically schedulable, we must rule out all dependence cycles. Recall that to handle recursive grammars, we allow cycles due to back-edges in the grammar graph.

We distinguish *back-edge* connections, which stem from back-edges in the grammar graph, from *normal* connections, which do not. Let $N \subseteq C$ be the subset of normal connections. For each attribute x in V, we use one integer l_x to impose a total order on all attributes while ignoring all back-edge connections.

$$\phi_{\text{Sched}} \stackrel{\text{def}}{=} \bigwedge_{(A,B)_c \in N} \left(m_c = \downarrow \Rightarrow \bigwedge_{x \in B} l_x > \max_{y \in A}(l_y)\right) \wedge \bigwedge_{(A,B)_c \in N} \left(m_c = \uparrow \Rightarrow \bigwedge_{x \in A} l_x > \max_{y \in B}(l_y)\right) \wedge \bigwedge_{(\alpha,V,I,S)_f \in F} \left(I \supset \varnothing \wedge e_f \Rightarrow \bigwedge_{x \in V \setminus I} l_x > \max_{y \in I}(l_y)\right)$$

Spanning The spanning property guarantees that all attributes are computed exactly once. In essence, we require that our $\mathcal{L}$-solvers be single-assignment AGs. However, in $H_{\mathcal{G}}$, some attributes might be assigned twice due to our representation of grammar recursion with back-edges.

Every attribute $x \in V$ must be computed by exactly one of the following: (i) a local solver synthesized from a node block (ψ_{F}); or (ii) a normal connection ($\psi_{\downarrow}$ and $\psi_{\uparrow}$). Some attributes might be additionally assigned by back-edge connections. In such cases, ϕ_{Conn} guarantees that all connections assigning to the same attribute share the same mode.

$$\phi_{\text{Span}}(x) \stackrel{\text{def}}{=} \bigodot \{\psi_{\text{F}}(x) \cup \psi_{\downarrow}(x) \cup \psi_{\uparrow}(x)\}$$

We define the logical connective $\odot$ to be true iff exactly one of its clauses is true.

$$\psi_{\text{F}}(x) \stackrel{\text{def}}{=} \{e_f \mid (\alpha, V, I, S) \in F \wedge x \in V \setminus I\}$$
$$\psi_{\downarrow}(x) \stackrel{\text{def}}{=} \{m_c = \downarrow \mid (A, B)_c \in N \wedge x \in B\}$$
$$\psi_{\uparrow}(x) \stackrel{\text{def}}{=} \{m_c = \uparrow \mid (A, B)_c \in N \wedge x \in A\}$$

$\mathcal{L}$-Solvers The translation of models of ϕ to functional AGs is straightforward: The e_f booleans indicate which local solvers to use.

Theorem 3 (Correctness)**.** *All models of ϕ are $\mathcal{L}$-solvers.*

6.3 Completeness

GM synthesis is incomplete and thus might fail to find a global solver, even when one exists. In Section 7, we show that GM synthesis is sufficiently complete in practice.

Recall that GM synthesis relies on the following hypothesis: the global solver is expressible as compositions of local solvers. The granularity of the decomposition affects whether our hypothesis holds. Coarser initial decompositions (*i.e.*, blocks) yield more local solvers at the expense of creating larger local synthesis problems, thus decreasing efficiency. We call the loss of completeness due to decomposition the *cost of modularity*, to distinguish it from the loss of completeness incurred due to any incompleteness of π.

We state a condition for hierarchical linear systems of equations sufficient to guarantee zero cost of modularity. For clarity, we consider a single document; the condition is generalizable by induction on the document grammar. Let the system $rel(d)$ be represented by the matrix of coefficients M_d. The decomposition of $rel(d)$ into blocks corresponds to a partition of the rows of M_d.

Theorem 4 (Completeness Condition)**.** *For linear equations, GM synthesis has no cost of modularity if M_d can be triangularized using row combinations (i.e., adding a linear combination of rows to another) only between rows belonging to the same block and row interchanges for any pair of rows.*

7. Evaluation

We evaluated GM synthesis along three axes:

- *Scalability vs. Completeness* GM synthesis trades completeness for scalability: Is it both scalable and sufficiently complete to synthesize $\mathcal{L}$-solvers for realistic DSLLs?
- *Performance* How does the solving speed of the synthesized $\mathcal{L}$-solvers compare with the speed of state-of-the-art, general-purpose constraint solvers?
- *Parameterizable* DSLL*s* Can our layout specifications produce $\mathcal{L}$-solvers for a diverse range of user interactions (*e.g.*, screen resizing; data updates), each updating different input variables?

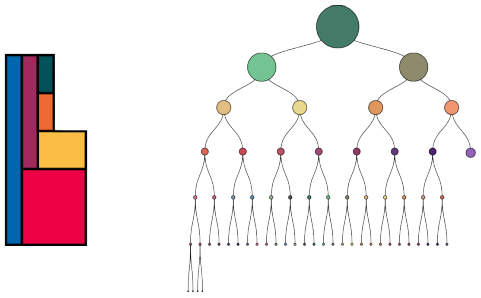
Experimental Setup GM synthesis is parametrized by the local synthesis procedure π. In our experiments, we implemented π with a combination of well-known techniques: For linear relations, we used a CEGIS loop [1, 28] iteratively trying templates of the form *if* $(l_1 > 0)$ l_2 *else* l_3 where l_1, l_2, and l_3 are linear combinations of attributes. Synthesis and verification queries are encoded in SMT linear real arithmetic [7]; attribute values are not bounded. Polynomial equations are solved in isolation, using a set of algebraic rewrite rules. Other tools could be used to implement π, for example Comfusy [18] and Sketch [28].

We used the Superconductor AG scheduler [23] to compile $\mathcal{L}$-solvers to (sequential) tree traversals. The resulting traversals are implemented in JavaScript and operate directly on the browser DOM. As a result, our custom $\mathcal{L}$-solvers can easily be deployed in any web browser. Figures 1b and 1c have been laid out by one of our $\mathcal{L}$-solvers. All of our benchmarks were run on a 2.5GHz Intel Sandy Bridge processor with 8GB of RAM using Firefox 30.0.

Case Studies To show that GM synthesis is widely applicable, we evaluated it on three DSLLs constructed with PBM [14], one for each of the three major layout domains:

(i) document (webpage) layout; (ii) GUI; and (iii) data visualization. Each language is full-fledged in that it computes all attributes required by rendering.

1. Our first case study is a guillotine language where a set of horizontal and vertical dividers partition the space (Figure 9a). Such a language can encode a subset of CSS [27]. The guillotine language totals 30 linear constraints and satisfies the completeness condition (Theorem 4).
2. Our second case study is a language of flexible grids [13]. Such languages are frequently used to layout widgets in graphical user interfaces [15] and some data visualizations. Figure 9b shows a tree layout based on flexible grids. The sizes of each cell of the grid are allocated based on a weighted sum. The weight of each cell is a runtime input, which produces non-linear constraints. The grid language consists of 47 constraints.
3. Finally, a language of treemaps [16], a visualization of hierarchical datasets popular in finance. The screen is tiled recursively, based on the area occupied by each subtree of the document (Figure 1). Each leaf has a runtime input corresponding to its relative area. Constraints involving area computations are non-linear. The treemap language has 40 constraints.



(a) Guillotine (b) Flexible grids

Figure 9. A guillotine layout (a) and a tree made with flexible grids (b). Both layouts were designed within the PBM system [14] and laid out by our synthesized $\mathcal{L}$-solvers.

To illustrate our DSLL specification language, we show below a partial definition of the treemap block that tiles the space horizontally. Lines 2 and 3 set up the relative coordinates *left*, *right*, denoting the relative horizontal displacement from the parent node, based on the absolute coordinate *x*. The third constraint is key: it binds the visual area of each document node to the value of the tile. These constraints are local in that they refer to parent and children in the document.

```
1 block hdiv(...) {
2     x == parent.x + left
3     left + width == right
4     scale * value == height * width
5     child1.left >= child0.width...
6 }
```

Scalability and Completeness Our GM synthesizer is sufficiently complete to successfully generate an $\mathcal{L}$-solver for each of the three case studies. Synthesis took less than four minutes in each case, an acceptable compilation time, with the local synthesis and the recomposition steps taking approximately equal time. Table 1 illustrates the complexity of the $\mathcal{L}$-solvers obtained after scheduling. Listed are the number of tree traversals, the number of local solvers used, as well as the size of the JavaScript code. For comparison, Mozilla's new Servo browser employs four passes to lay out CSS [4]. The number of local solvers is per grammar ($\mathcal{L}$-solver), rather than for a document. Finally, the number of lines of code reported includes only the solver itself (*i.e.*, the computation of document attributes); the rendering code has been excluded.

Language	Traversals	Local Solvers		SLOC	Time
		Total	Selected		
Guillotine	t	289	74	317	126
Grid	$t\,;b\,;t$	385	89	483	175
Treemap	$t\,;b\,;t\,;b\,;t$	394	91	599	194

Table 1. The complexity of our $\mathcal{L}$-solvers. The second column shows the number and type of tree passes: t and b denote top-down and bottom-up passes, respectively. The third column reports the number of local solvers synthesized and the subset used. The last two columns show the number of lines of code and the total synthesis time in second.

GM synthesis provides no guarantee that the $\mathcal{L}$-solvers are optimal, neither in the number of traversals nor in the number of operations. We manually checked each $\mathcal{L}$-solver: all are optimal in the number of traversals.

We also compare GM synthesis with non-modular functional synthesis methods, specifically with Comfusy and Sketch. These techniques are limited to synthesis of d-solvers (*i.e.*, they do not generalize to languages of documents), hence we asked them to synthesize a solver for a single document of 127 nodes. Both systems failed to synthesize a d-solver in less than one hour. These results suggest that GM synthesis may strike a good balance between completeness and scalability in the domain of layout engines.

Performance We compare the performance of our $\mathcal{L}$-solvers with Z3 [10], a state-of-the-art constraint solver. Our solvers are implemented in JavaScript, a relatively slow language. Z3 solves the constraint system defined by the document ($rel(d)$) at runtime. We tested several solver algorithms implemented in Z3.

We measured the time to compute the layout of documents from 255 to 16383 nodes, for each of our DSLLs. Such document sizes are typical [29]. For reference, the front page of `nytimes.com` contains over 3000 nodes and data visualizations tend to be larger. Our benchmark documents are balanced trees generated randomly. For Z3, we chose the fastest SMT theory which could express the layout specification. Interestingly, the non-linear real arithmetic

solver was faster than 16bit bitvectors for both the grid and treemap languages. For guillotine, we used linear real arithmetic. Table 2 summarizes our results.

Doc Size	Guillotine		Grid		Treemap	
	GM	Z3	GM	Z3	GM	Z3
255	3	705	5	707	8	680
1023	10	2310	19	1494	49	1935
4095	41	12800	81	8403	120	8935
16383	162	>3 min	213	—	261	—

Table 2. Time to compute the layout in millisecond. Missing entries (—) indicate "unknown" answers (no model). Notice that our $\mathcal{L}$-solvers scale linearly with the document size.

Our $\mathcal{L}$-solvers scale linearly with the size of the document, whereas Z3 fails on the largest document (either timing out or reporting "unknown") for all three case studies. This speedup is explained by GM synthesis moving the backtracking search performed at runtime by Z3 to compile time, leaving only function applications to runtime. On the medium-sized document (1023 nodes), $\mathcal{L}$-solvers are between 39 and 231 times faster than Z3. Our results show that across the three case studies, our $\mathcal{L}$-solvers are fast enough (<0.5 second) for interactive settings.

***Parameterizable* DSLLs** We illustrate the expressiveness of non-directional constraints by synthesizing multiple $\mathcal{L}$-solvers from the same DSLL, each parameterized by a different set of runtime inputs. Each solver recomputes the layout in response to some event triggered by a user interaction. Each event sets the values of some runtime inputs, from which all remaining attributes are computed.

We illustrate the expressive power of non-directional constraints on the language of treemaps. Assume that the treemap represents the market capitalization of some companies. The leaves of the document are companies, while inner nodes encode the tiling of the screen (Figure 1a). We consider three events: (i) the values of all companies are updated; (ii) the user resizes the treemap; and (iii) the user moves the treemap.

For the first event, the set of runtime inputs is the *value* attribute of each company (*i.e.*, leaf nodes). Given new values, the layout engine must update the sizes of each node, including the overall size of the treemap (root node). In contrast, the second event updates the overall size of the treemap. As such the runtime inputs are the height/width of the root node. The values of leaves remain unchanged, and the layout engine must recompute the scaling parameter converting values (dollars) into areas (squared pixels). From our treemap DSLL, our synthesizer generates three $\mathcal{L}$-solvers, one per set of runtime inputs (Table 3). Note that the $\mathcal{L}$-solvers are dramatically different from each other in that they require a different number of tree passes.

To conclude, we have shown empirically that (i) GM synthesis is both scalable and complete enough to generate $\mathcal{L}$-solvers for a variety of DSLLs; (ii) our $\mathcal{L}$-solvers outperform general-purpose constraint solvers; and (iii) relational AGs are a concise formalism for expressing interactive layouts with multiple flows of computation.

Event	Inputs	Traversals	SLOC
New market cap.	$t.value \; \forall$ tiles t	5	599
Resize root	*root.w*, *root.h*	3	463
Reposition root	*root.x*, *root.y*	1	165

Table 3. Three $\mathcal{L}$-solvers, one per event, recomputing the treemap layout from different sets of inputs. For space reasons, we only show input attributes with new values and omit those which are input but remain constant. We report the number of tree-traversals and the size of code.

8. Related Work

GM synthesis builds upon previous work in program synthesis. Our work is closely related to constraint planning, mode inference in attribute grammars, and logic programming.

Program Synthesis Functional synthesis, a subset of program synthesis [20, 21], is an instance of the AE-paradigm, also known as the Skolem paradigm for synthesis [24]. GM synthesis builds upon functional synthesis procedures, such as Comfusy [18] or Sketch [28], by enabling modular decompositions of specifications to gain scalability. Singh et al. [26] also propose a modular synthesis technique but use a different form of modularity.

Constraint Planning (CP) The task of finding a d-solver can be cast as a multi-way (*i.e.* non-directional) constraint planning problem, for which solvers like SkyBlue [25] and QuickPlan [31] have been proposed. In CP, each "planning constraint" corresponds to a block in our framework. Like our d-solver setting, given a set of planning constraints, each associated with local solvers (methods), a planner finds a sufficient subset of functions that computes all attributes. In contrast with our approach, a programmer is responsible for providing enough local solvers as well as partitioning the specification into blocks, to satisfy special requirements of the algorithm. QuickPlan works in quadratic-time by imposing a clever restriction on planning constraints: each local solver must mention all variables of its planning constraint, either as input or as output. The programmer satisfies this restriction by intelligently factoring clauses of the specification into planning constraints when writing local solvers. In our setting, the same information is left to the oracle (*i.e.*, we search over the space of all block decompositions). As illustrated in Section 4.1, our oracle decomposes node blocks into smaller blocks, each corresponding to one planning constraint. Without this step, we would be restricted to computing all attributes of each node block with a single local solver, which would prevent creating global solvers for documents requiring multiple tree passes. In essence, we cannot use QuickPlan to compute d-solvers, because we do not know

upfront how many passes are needed. In practice, we synthesize local solvers for all block decompositions. As a result, we obtain many more local solvers than in the traditional constraint planning setting. Naively encapsulating local solvers into planning constraints meeting QuickPlan's simplifying assumption would create an exponential explosion. With one planning constraint per decomposed block, QuickPlan's complexity would become $(2^n)^2$ where n is the number of clauses in the specification. In general, constraint planning for non-directional constraints is NP-complete [19].

We distinguish ourselves by supporting not only finite relations but also tree grammars of relations, enabling the same $\mathcal{L}$-solver to lay out multiple documents (datasets), while still guaranteeing a static schedule.

Attribute Grammar Our modular synthesis algorithm has close connections with relational AGs and logic programming. Deransart and Maluszynski [12] give theoretic constructions demonstrating how relational grammars, functional grammars and directed clause programs are related to one another. Mode analysis [11] techniques for logic programs, which compute whether clause arguments of logical programs are input or output, could be—in principle—transposed to AGs to compute whether attributes are inherited or synthesized. The principal goal of mode inference is to learn static properties enabling compiler optimizations. To this end, such techniques rely on abstract domains to soundly perform over-approximations of modes. Our work differs in two ways. First, to obtain executable $\mathcal{L}$-solvers, we must compute exact modes for all attributes. As such, we cannot apply techniques trading precision for scalability or termination. Secondly, our approach is modular. For each block, we synthesize a set of local solvers, which can be viewed as sets of possible modes for a block. Local solvers are computed independently for each block and can be reused across DSLLs. Mode analysis techniques based on abstract interpretation operate on the whole program.

Constraint Logic Programming (CLP) In CLP [2, 3, 33], constraint systems are flat and unstructured while we exploit the tree structure to produce $\mathcal{L}$-solvers in a modular fashion. Furthermore, given a relational specification of a document, CLP tools search for one layout (*i.e.*, solution) among the potentially many, whereas we ensure that the specification is functional. That is, the layout is uniquely determined by known attributes (*i.e.*, deterministic).

9. Conclusion

We presented grammar-modular synthesis, a new algorithm exploiting the structure of hierarchical specifications to scale synthesis to large relations at the cost of completeness. We synthesized tailored layout solvers for custom languages of documents. Our three case studies show not only that GM synthesis scales to large specifications which could not be tackled by state-of-the-art tools, but also that the $\mathcal{L}$-solvers generated outperform general-purpose constraint solvers by one order of magnitude. For our domain, layout, we believe that GM synthesis strikes the right balance between scalability of synthesis, completeness of synthesis, and performance of the resulting $\mathcal{L}$-solvers.

We are interested in applying GM synthesis to domains beyond document layout. For instance, the techniques presented in this paper could potentially generate parametrized hardware designs from trees of components with constraints such as size or delay.

Acknowledgments

This work was supported in part by NSF Grants CCF–1018729, CCF–1139138, CCF–1337415, and CCF–0916351, a Grant from DOE FOA–0000619, and a grant from DARPA FA8750–14–C–0011, as well as gifts from Mozilla, Nokia, Intel and Google.

References

[1] Alur, R., Bodik, R., Juniwal, G., Martin, M. M. K., Raghothaman, M., Seshia, S. A., Singh, R., Solar-Lezama, A., Torlak, E., and Udupa, A. (2013). Syntax-guided synthesis. In *FMCAD*, pages 1–17. IEEE.

[2] Apt, K. (2003). *Principles of Constraint Programming*. Cambridge University Press, New York, NY, USA.

[3] Apt, K. R. and Wallace, M. (2007). *Constraint Logic Programming Using Eclipse*. Cambridge University Press, New York, NY, USA.

[4] Atkinson, E. (2015). Personal communication.

[5] Ausiello, G. (1988). Directed hypergraphs: Data structures and applications. In Dauchet, M. and Nivat, M., editors, *CAAP '88*, volume 299 of *Lecture Notes in Computer Science*, pages 295–303. Springer Berlin Heidelberg.

[6] Bachrach, J., Vo, H., Richards, B., Lee, Y., Waterman, A., Avižienis, R., Wawrzynek, J., and Asanović, K. (2012). Chisel: Constructing hardware in a scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*, DAC '12, pages 1216–1225, New York, NY, USA. ACM.

[7] Barrett, C., Stump, A., and Tinelli, C. (2010). The Satisfiability Modulo Theories Library. `www.smt-lib.org`.

[8] Bos, B., Çelik, T., Hickson, I., and Lie, H. W. (2011). Css 2.1 spec. `www.w3.org/TR/CSS2/`.

[9] Comon, H., Dauchet, M., Gilleron, R., Löding, C., Jacquemard, F., Lugiez, D., Tison, S., and Tommasi, M. (2007). Tree automata techniques and applications. Available on: `www.grappa.univ-lille3.fr/tata`.

[10] De Moura, L. and Bjørner, N. (2008). Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg. Springer-Verlag.

[11] Debray, S. K. and Warren, D. S. (1988). Automatic mode inference for logic programs. *Journal of Logic Programming*, 5(3):207–229.

[12] Deransart, P. and Maluszynski, J. (1985). Relating logic programs and attribute grammars. *Journal of Logic Programming*, 2(2):119–155.

[13] Feiner, S. K. (1988). A grid-based approach to automating display layout. In *Proceedings on Graphics Interface '88*, pages 192–197, Toronto, Canada. Canadian Information Processing Society.

[14] Hottelier, T., Bodik, R., and Ryokai, K. (2014). Programming by manipulation for layout. In *Proceedings of the 27th Annual ACM Symposium on User Interface Software and Technology*, UIST' 14, pages 231–241, New York, NY, USA. ACM.

[15] Hurst, N., Li, W., and Marriott, K. (2009). Review of automatic document formatting. In *Proceedings of the 9th ACM Symposium on Document Engineering*, DocEng '09, pages 99–108, New York, NY, USA. ACM.

[16] Johnson, B. and Shneiderman, B. (1991). Tree-maps: A space-filling approach to the visualization of hierarchical information structures. In *Proceedings of the 2Nd Conference on Visualization '91*, VIS '91, pages 284–291, Los Alamitos, CA, USA. IEEE Computer Society Press.

[17] Knuth, D. E. (1968). Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145.

[18] Kuncak, V., Mayer, M., Piskac, R., and Suter, P. (2010). Complete functional synthesis. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 316–329, New York, NY, USA. ACM.

[19] Maloney, J. H. (1992). *Using Constraints for User Interface Construction*. PhD thesis, University of Washington, Seattle, WA, USA.

[20] Manna, Z. and Waldinger, R. (1980). A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 2(1):90–121.

[21] Manna, Z. and Waldinger, R. J. (1971). Toward automatic program synthesis. *Communications of the ACM*, 14(3):151–165.

[22] Meyerovich, L. A. and Bodik, R. (2010). Fast and parallel webpage layout. In *Proceedings of the 19th International Conference on World Wide Web*, WWW '10, pages 711–720, New York, NY, USA. ACM.

[23] Meyerovich, L. A., Torok, M. E., Atkinson, E., and Bodik, R. (2013). Parallel schedule synthesis for attribute grammars. In *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '13, pages 187–196, New York, NY, USA. ACM.

[24] Pnueli, A. and Rosner, R. (1989). On the synthesis of a reactive module. In *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '89, pages 179–190, New York, NY, USA. ACM.

[25] Sannella, M. (1994). Skyblue: A multi-way local propagation constraint solver for user interface construction. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology*, UIST '94, pages 137–146, New York, NY, USA. ACM.

[26] Singh, R., Singh, R., Xu, Z., Krosnick, R., and Solar-Lezama, A. (2014). Modular synthesis of sketches using models. In *Verification, Model Checking, and Abstract Interpretation*, volume 8318 of *Lecture Notes in Computer Science*, pages 395–414. Springer Berlin Heidelberg.

[27] Sinha, N. and Karim, R. (2013). Compiling mockups to flexible uis. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 312–322, New York, NY, USA. ACM.

[28] Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., and Saraswat, V. (2006). Combinatorial sketching for finite programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 404–415, New York, NY, USA. ACM.

[29] Souders, S. (2013). How fast are we going now? `www.stevesouders.com/blog/2013/05/09/how-fast-are-we-going-now/`.

[30] Swierstra, S. D., Azero Alcocer, P. R., and Saraiva, J. (1999). Designing and implementing combinator languages. In *Advanced Functional Programming*, volume 1608 of *Lecture Notes in Computer Science*, pages 150–206. Springer Berlin Heidelberg.

[31] Vander Zanden, B. (1996). An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 18(1):30–72.

[32] Warme, D. M. (1998). *Spanning Trees in Hypergraphs with Applications to Steiner Trees*. PhD thesis, University of Virginia, Charlottesville, VA, USA.

[33] Yap, R. H. C. (2004). Constraint processing. *Theory and Practice of Logic Programming*, 4(5-6):755–757.