Program Synthesis for Hierarchical Specifications

Thibaud Hottelier
Ras Bodik

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-139
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-139.html

July 29, 2014

Acknowledgement

This work was supported in part by awards from the National Science Foundation (CCF–0916351, CCF–1139138, and CCF–1337415), as well as gifts from Google, Intel, Mozilla, Nokia, and Samsung.

Program Synthesis for Hierarchical Specifications

Thibaud Hottelier
UC Berkeley
tbh@cs.berkeley.edu

Ras Bodik
UC Berkeley
bodik@cs.berkeley.edu

Abstract

Synthesis is the problem of obtaining programs from relational specifications. We present grammar-modular (GM) synthesis, an algorithm for synthesis from tree-structured relational specifications. GM synthesis makes synthesis applicable to previously intractable relational specifications by decomposing them into smaller subproblems, which can be tackled in isolation by off-the-shelf synthesis procedures. The program fragments thus generated are subsequently composed to form a program satisfying the overall specification. We also generalize our technique to tree languages of relational specifications. Here, we synthesize a single program capable satisfying any (tree-shaped) relation belonging to the language; the synthesized program is syntax-directed by the structure of the relation. We evaluate our work by applying GM synthesis to document layout: given the semantics of a layout language, we synthesize tailored constraint solvers capable of laying out languages of documents. Our experiments show that GM synthesis is sufficiently complete to successfully generate solvers for non-trivial data visualizations, and that our synthesized solvers are between 39- to 200-times faster than general-purpose constraint solvers.

1. Introduction

By raising the level of abstraction, automatic synthesis of programs from specifications has the potential to make programming easier. Program specifications can often be stated as a relation between inputs and outputs, for instance, with pre and post conditions. Then we can synthesize a program by turning the relation (specification) into a function (program) according to an input/output (known/unknown) partition of the relation's variables. This type of program synthesis is usually called functional synthesis. In this paper, we focus on relations expressible in propositional SMT logics.

A key benefit of functional synthesis is enabling programmers to use declarative constraints (*i.e.*, relations) without incurring the cost of solving constraints. Generally, constraints are computed at runtime by a solver for a particular input (*i.e.*, initial conditions). By synthesizing functions from constraints, we obviate the need for constraint solving at runtime and shift this cost to compilation time. Intuitively, the synthesized functions execute only value propagations and bypass the backtracking search performed by constraint solvers. In situations where the same constraint system must be solved multiple times with varying inputs, the same synthesized functions can be reused, making the performance gains brought by synthesis even more attractive.

Our goal is to scale functional synthesis to large relations. At heart, functional synthesis is a quantifier elimination problem: we eliminate variables from the relation until all outputs can be expressed only in terms of inputs. Recent work such as Comfusy [15] has brought program synthesis to mainstream compilers. Comfusy is an extension of Scala allowing relational constraints in functional programs. In essence, Comfusy translates the execution of a quantifier-elimination procedure on a particular relation into SMT formula whose models capture the key steps of quantifier elimination. Ultimately, given such steps, Comfusy can construct functions computing the outputs. In fact, such functions can be viewed as specialized solvers tailored for one particular constraint system. In practice, the efficiency of the translation and scalability of SMT solvers limit the size of relations which can be functionalized, *i.e.*, for which we can compute (executable) functions. Empirically, we found that quantifier-elimination based approaches do not scale to the large specifications of our domain, document and visualization layout.

Synthesis of Layout Engines Layout specifications are naturally expressed with constraints [12]. Constraint-based languages such as CCSS [4] and ALE [31] are both powerful and versatile. Even CSS, the ubiquitous web template language, relies on constraints, although in a more restricted and indirect manner [12]. Constraints also enable the inference of layout specifications directly from user demonstrations [11, 22].

Layout engines compute attributes such as the sizes and positions of all visual elements from input attributes, which are runtime constants (*e.g.*, the window size). When layout is specified with constraints, solving them quickly enough (less than half a second) to enable smooth user interactions is a major technical challenge. Today, the average webpage has over a thousand elements, each with dozens of attributes [27]. Since layout engines are executed numerous times, for instance, to handle resize-window events or to adapt to new data values, the potential cumulative runtime savings from synthesizing specialized "function" solvers are large. For

these reasons, we believe that automatic generation of layout engines is a prime target for functional synthesis.

Layout can be computed either by solving constraints at runtime (*i.e.*, after input values are provided) with a general-purpose solver, or by handcrafting a solver (engine) tailored to a particular type of visualization (D3 [5]). Handcrafted solvers are usually implemented as a bounded set of tree traversals over a hierarchical document labeled with constraints. Today, general-purpose solvers are too slow for interactive settings (up to 200x slower than handcrafted solvers, see Section 4). As such, all browsers and most visualization libraries (*e.g.*, D3 and Protovis [10]) rely on handcrafted solvers. However, writing such solvers is time-consuming. As a result, trying out design ideas is expensive.

Functional synthesis promises to combine the performance of handcrafted engines with the ease of use of constraint solvers. By applying functional synthesis to the relational specification of layouts, we generate a solver specialized for a particular set of constraints. In essence, we automate the tedious optimizations currently performed manually by visualization programmers. Ultimately, layout is a domain in which functional synthesis could have a significant impact.

However, scaling synthesis to large relations is a challenge; so far, synthesis has mostly been limited to producing program fragments. Our experiments show that Comfusy scales to 100 variables at most. However, the relations describing layout can range over 10^4 program variables, more than one order of magnitude larger than what state-of-the-art synthesis tools can handle. We present grammar-modular (GM) synthesis, a technique to scale functional synthesis to large and hierarchical relations, such as data-visualization specifications.

Modular Synthesis To scale synthesis to large relations, we rely on the presence of a hierarchical structure to trade completeness for scalability. Specifications can often be written as conjunctions of smaller relations. We exploit this structure to decompose the synthesis problem into smaller subproblems whose solutions can eventually be combined together to functionalize the overall relation. We call this technique *modular* synthesis.

Layout specifications, for instance, naturally give rise to a hierarchical decomposition. The data to be laid out is commonly represented as a tree of nodes — the document. Each document node is encoded as its own conjunct. We can apply synthesis on each node individually and then combine the results together to create an engine computing the layout for the whole document.

The key technical challenge of modular synthesis is the construction of a global function satisfying the overall relation, from the local functions produced by applying functional synthesis on each sub-relation. We cast the creation of a global function as (function) compositions of local functions. By doing so, we trade completeness for efficiency: modular synthesis cannot perform deduction across decomposition boundaries (only function composition), so a relation that can be functionalized globally may not be functionalizable in a modular way. The smaller relations may not be functional and hence the necessary local functions cannot be produced.

Grammar-Modular Synthesis So far we have outlined how modular synthesis can generate functions solving one particular relation. In the layout domain, each type of document node instantiates the same local constraints. For performance, we would like to avoid applying the GM synthesizer anew for each relation (*i.e.* document). We would like to avoid not only re-synthesis of local functions, but also the expensive composition of the global function, and simply thread local functions together based on the tree structure of the document.

Imagine you have written a specification of a simple visualization: a barchart. The synthesis techniques presented so far would generate an engine specific to this particular barchart. That is, the engine would only function on a single document. Our constraints bind a fixed set of variables; relations for barcharts with a different number of bars have a different number of variables. If the dataset changes to require more bars, a new engine must be synthesized.

To be practically useful, we must synthesize engines capable of adapting to such changes by handling multiple documents, each with a particular number of bars, for instance. That is, the engine must be generic enough to solve multiple relations, even an unbounded number of relations. Figures 1(b) and 1(c) show two documents from a language of treemaps laid out by the same synthesized engine.

In essence, we generalize program synthesis to accept not a fixed relation but a language of relational specifications. We restrict ourselves to *regular* tree-languages of relations whose variable sharing structure forms a tree. By doing so, we can represent the synthesized program as a set of functions whose composition is syntax-directed by the structure of the relation. Fundamentally, we are converting a relational attribute grammar[1] (AG) into a traditional, functional attribute grammar that is statically schedulable. We call this technique *grammar-modular* (GM) synthesis. Given a language of relations and an input/output partition of its variables, we synthesize a functional attribute grammar capable of computing the outputs of any relation in the language. More technically, we generalize modular synthesis to grammars of relations by handling alternative and recursive productions. To guarantee that our functional attribute grammars are statically schedulable, we reject grammars with cyclic dependencies between attributes, thereby forbidding fixed-point computations.

Regular tree-languages include most layout languages, including data visualizations. GM synthesis enables automatic generation of layout engines from specifications of layout languages. Such languages define both syntactically legal documents and their layout semantics. Each such grammar

[1] A relational AG is an AG with constraints (*i.e.*, relations) instead of update functions [8, 14].

defines a language of documents and its layout semantics. The layout engine, a functional attribute grammar, computes all document attributes (*e.g.*, sizes, position) given runtime-inputs (*e.g.*, window size), which are given as values of some attributes. Eventually, the layout engine can be scheduled to tree traversals with the same form as handcrafted engines. In fact, from the same relational specification, we can synthesize distinct layout engines, depending on which attributes are inputs and which attributes are computed, which is useful in interactive situations. For instance, in a scroll-box, when the user moves the slider, the document position can be computed from the slider and vice versa. Each such user interaction triggers a different flow of attribute updates, but maintains the same constraints.

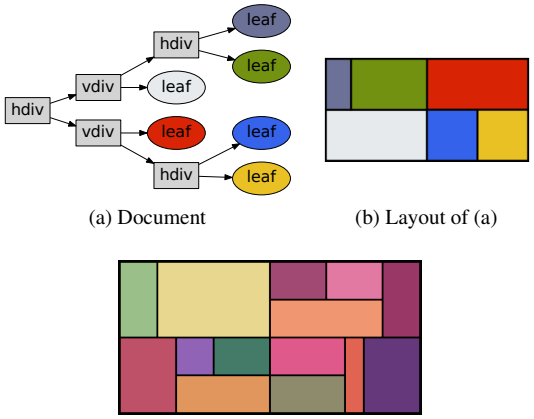
This paper makes the following contributions:

1. We present grammar-modular synthesis, a new technique enabling program synthesis to scale to previously intractable problems. We rely on the specification having a hierarchical structure to decompose it into smaller parts. Then we perform synthesis on each of them individually, and combine the resulting functions into a program satisfying the overall specification.
2. We apply GM synthesis to generation of layout engines, not only for single documents but for languages of documents. We demonstrate empirically that our synthesized engines are sufficiently complete and outperform Z3, a general-purpose constraint solver, by up to 200 times.
3. For constraints expressible as linear equations, we state a necessary and sufficient condition on the decomposition of the specification, guaranteeing the completeness of GM synthesis. We also define a class of constraints for which GM synthesis is complete independently of the decomposition.

2. Background and Motivation

In this section, we briefly introduce document layout, our application domain. We give an overview of our technique in Section 3.2. Even though the challenges posed by layout led us to develop GM synthesis, the techniques presented in this paper are generic. We use GM synthesis to compile high-level layout specifications to tailored layout engines but the algorithm is applicable to any hierarchical specification expressible as a relational attribute grammar; it could, perhaps, be used to raise the level of abstraction in compiler construction, which is often specified as a functional attribute grammar. Layout itself is already a vast domain which spans across graphical user interfaces, data-visualizations, and documents.

Documents and Blocks A *document* is a tree of *nodes*, each labeled with *blocks*. Blocks play the role of building "bricks". For instance, an image node and a text node wrapped under their parent paragraph node constitute a simple document. Blocks define the layout semantics of document nodes: typically, how each node's positions and sizes are computed. As such, the block of a node is akin to its "type". Formally, each block gives a visual appearance and attributes (*e.g.*, positions and sizes) to document nodes. Some attributes can be marked as *input*; these are runtime constants unknown at compile time, *e.g.* the size of an image, or the size of the top-level window. Solving the layout of a document amounts to computing the values of all attributes given input values, in accordance with the blocks' semantics.



(a) Document (b) Layout of (a)

(c) A larger treemap

Figure 1. A treemap visualization. A tree of block-labeled nodes constituting a document (a) together with its layout (b). Figure (c) shows the layout of a larger document constructed from the same blocks and belonging to the same language (Listing 2).

Figure 1 shows a document and its layout for a treemap, a popular visualization often used to compare companies' relative market valuations[2]. The document is constructed from nodes that use four kinds of blocks: Document leaves (*leaf*) represent companies; their value (market valuation) is an input. Leaves are the only nodes with a visual appearance. The invisible inner nodes (*h/vdiv*) enforce the recursive division of space; *i.e.* they position their children. Finally, the document *root* (not shown in Figure 1(a)) is labeled with a special *root* block. We show below some of the constraints behind the *hdiv* block. We discuss our layout constraints at the end of this section.

```
1 block hdiv(...) {
2   x = parent.x + left
3   left + width = right
4   scale * value = height * width
5   width = child0.width + child1.width
6   child1.left >= child0.width
7   ...
8 }
```

Listing 1. The constraints defining the *hdiv* block of the treemap. Constraints refer to attributes of parent and child

[2] See SmartMoney's Map of the Market on `www.marketwatch.com`.

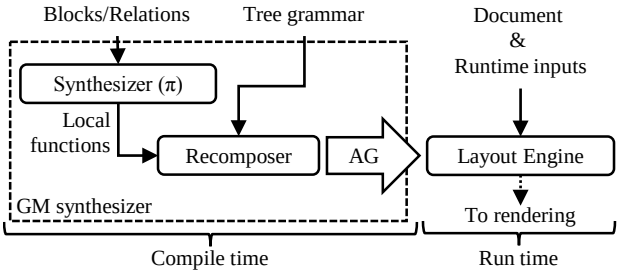


Figure 2. The architecture of our GM synthesizer. The first step of GM synthesis — decomposition — is not shown. The attribute grammar scheduler is out of the scope of this paper. Its output is the layout engine itself.

nodes in the document tree. Equal symbols denote equalities, not assignments.

Synthesis of Layout Engines A layout specification consists of two parts: (i) a definition of the layout semantics of each block; and (ii) a description of which nestings of nodes are allowed in documents. Together, both parts constitute a relational attribute grammar [14] which defines a language of documents together with layout semantics.

We specify legal nestings of blocks with regular tree-grammars[3]. Each terminal of the tree-grammar corresponds to a block-labeled node. Consequently, every derivable tree is a document. We show below our grammar of treemaps. Non-terminals are capital letters. The document of Figure 1(a) belongs to this grammar. Note that we enforce the alternation of horizontal and vertical splits (*h/vdiv*).

```
S ::= root(H)
H ::= hdiv(V, V)      V ::= vdiv(H, H)
   |  leaf()             |  leaf()
```

Listing 2. The tree-grammar of our treemap language.

Given a set of blocks and a tree-grammar, our synthesizer outputs a layout engine, in the form of a functional attribute grammar [14], capable of computing the layout of all derivable documents (Figure 2). We guarantee that the resulting functional attribute grammars are always statically schedulable. Such attribute grammars are compilable to efficient tree traversals [19]. In contrast with the backtracking search employed by general-purpose constraint solvers, our layout engines perform only value propagations and function applications. The search happens at synthesis (compile) time. Assuming deterministic specifications, our synthesized engines always compute the same layout as general-purpose constraints solvers. Our language of constraints and our GM synthesizer are mature enough to be used in programmability user studies.

In the next paragraph, we detail how we define the layout semantics of blocks and explain why we believe relational attribute grammars are a suitable formalism for layout specifications.

Blocks Semantics Each block defines a set of attributes; these attributes decorate document nodes. The semantics of blocks are defined by placing constraints on attributes.

Constraints are *local* — only attributes from the direct parent or children (in the document hierarchy) can be referred to. We represent such references indirectly, by introducing equality constraints (Section 3.1).

Constraints are also *non-directional* — they leave the flow of computation unspecified, up to the synthesizer. Non-directionality raises the level of abstraction, enabling the same specification to capture alternative flows of computation. In interactive visualizations, values may flow in either direction, depending the user's actions. User actions change the value of an attribute, which is then considered an input. For instance, in a scroll-box, dragging the slider changes the slide's position, from which the new viewport position is computed. Alternatively, growing the viewport content updates the slider position. In this scenario, the same constraints are maintained (the slider position reflects the viewport position) but the set of known (*i.e.*, input) attributes differs. For each such interaction, the layout is computed from different input attributes, producing distinct flows of computation. Non-directionality lets us synthesize one layout engine for each interaction (*i.e.*, each set of input attributes) from the same specification. As such, interactive data visualizations can be captured very concisely.

To outline our constraint language, we show a simplified version of one block (*hdiv*) of our treemap language in Listing 1. Lines 2&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 group of companies represented.

The techniques presented in this paper are independent of the logical theories used to express constraints. Our examples and implementation rely on polynomial equations and linear inequalities over rationals, augmented with basic trigonometric functions as well as min/max operators. Empirically, we found such constraints expressive enough to specify a wide class of layouts and visualizations.

This paper is organized as follows: In Section 3, we start by introducing GM synthesis for single relations, then generalize our algorithms to grammars of relations, and finally discuss the completeness of our approach. Finally, we present our experimental results on synthesis of layout engines (Section 4) and discuss related work (Section 5).

3. Grammar Modular Synthesis

In this section, we first formalize concepts introduced previously and then present GM synthesis applied to layout engines. To simplify the presentation, we start by explaining our

[3] Regular tree-languages can be viewed as the set of derivation trees of a context-free word grammar [6].

technique on a language containing a single document (*i.e.*, the grammar has a single derivation). In a second step, we generalize our approach to languages of documents. Finally, we discuss the completeness of GM synthesis.

3.1 Preliminaries

For the sake of readability, we introduce the following notations: Let $f : D^m \rightarrow D^n$ be a function computing n variables given m variables, all in domain D. We denote by $\hat{f}[I, O]$ the function f lifted to symbolic variables, where I is the list of variables read ($|I| = m$), and O is the list of variables computed ($|O| = n$). For example, if $f(x_1, x_2) \stackrel{\text{def}}{=} (x_1 + x_2, 2x_1 - x_2)$ then $\hat{f}\,[\{a, b\}, \{c, d\}]$ represents $c := a + b$ and $d := 2a - b$. We purposely abstract away the mapping of variables onto arguments. Similarly, for relations of arity $m + n$, for instance $R(x_1, x_2, x_3) \stackrel{\text{def}}{=} x_1 + x_2 = x_3$, we write $(a, b, c) \in \hat{R}$ to denote $a + b = c$. For convenience, we extend our notation to lists of variables and write $O = \hat{f}(I)$ and $(I \cup O) \in \hat{R}$. In the context of layout, variables range over $\mathbb{Q}$ and are called *attributes*.

Functional Synthesis The functional synthesis problem is to find a total function f given a relation R and a partition of its variables into input/output lists I, O, respectively, such that $(I \cup \hat{f}(I)) \in \hat{R}$ for valuations of I. Such a function exists if R is functional in I. That is, $(I \cup O) \in \hat{R} \wedge (I \cup \hat{f}(I)) \in \hat{R} \implies O = \hat{f}(I)$ holds. As such, f is semantically unique (but may have multiple implementations). For convenience, we say that *f functionalizes R* with respect to inputs I.

We write $\pi_{I,O}(R)$ to denote the procedure finding such a function; the procedure fails if the function does not exist. GM synthesis relies on a functional synthesizer (π) to perform synthesis locally, on the subproblems created by decomposing the specification. π can be implemented using existing techniques (see Section 4).

Blocks and Documents We start with definitions of blocks and documents.

Definition 1 (Block)**.** *A* block *is a pair (V, R), where V is a finite set of attributes and R is a relation over V. Some attributes of V can be marked as* inputs*,* i.e. *runtime constants. The relation R is the conjunction of the constraints defining the layout semantics of the block. We assume that R is in CNF. That is, R is a conjunction of clauses $cl_0 \wedge \ldots \wedge cl_n$.*

Definition 2 (Document)**.** *A* document *is a tree of block-labeled nodes. Each document node contains the attributes and the relation of its block. As such, a block acts as the "type" of a node and through the layout constraints in the relation, the block establishes its layout semantics.*

To represent semantic connections between document nodes, we place additional equality constraints between attributes from a parent and its children (in the tree hierarchy). Formally, a connection c, denoted by $(A, B)_c$, is an equality constraint between the sets of attributes A and B. For now, both A and B are singleton sets.

Finally, given a document d, let I_d be the set of attributes of d marked as input. Let O_d be all other (non-input) attributes of d. Let $rel(d)$ be the relation representing the underlying constraint system of d. Formally, $rel(d)$ is the conjunction of the the relation of every document node as well as the equality constraints stemming from connections between nodes.

Definition 3 (d-Solver)**.** *Given a document d, a* d-solver *is a function $f\,[I_d, O_d]$ which functionalizes $rel(d)$.*

Modular Synthesis To synthesize a d-solver for a particular document, the simplest approach would be to use π directly and compute $\pi_{I_d,O_d}(rel(d))$. This is impractical in practice for all but the most trivial documents, since $rel(d)$ may be large and have more than a thousand of attributes. Consequently, we need a way to divide d-solver synthesis into simpler, independent subproblems. Our approach relies on the following hypothesis: the d-solver can be expressed as a composition of smaller, "local" functions, synthesized from each subproblem individually.

Given a document d, we synthesize a d-solver in three steps: (i) we decompose the specification ($rel(d)$) into conjuncts; (ii) we perform synthesis locally, on each individual conjunct, thus obtaining *local* functions; and (iii) we select and compose just enough local functions to construct a *global* function computing all attributes of d, thus creating a d-solver. Before we detail each of the three steps, we highlight the algorithmic challenges by constructing a d-solver for a small document with the help of an oracle.

3.2 Example

Let us consider a document comprised of two nodes labeled with block $a \stackrel{\text{def}}{=} (V_a, R_a)$ and block $b \stackrel{\text{def}}{=} (V_b, R_b)$, respectively. The specification of each block is shown below:

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

Our document has one input, denoted by attribute i. For the sake of the explanation, we abstract away connections. Instead, our two nodes directly share connected attributes. Here, both nodes share attributes x and y. As such, the specification of the document — $rel(d)$ — is simply $R_a \wedge R_b$. To create a d-solver, we must synthesize a function computing attributes $O_d = \{x, y, z\}$ from input attribute $I_d = \{i\}$.

Decomposition (Step 1) The first step is to decompose $rel(d)$. We follow the document structure and create two subproblems, one per node of the document.

Local Synthesis (Step 2) The second step consists of generating local functions for each node of the document. First, we ask the oracle to partition each block relation into subsets of clauses. Intuitively, each subset corresponds to one "pass" of the global function through the corresponding block. Then

we ask the oracle to partition the attributes of each block into input/output sets. Finally, we synthesize local functions for each of set of clauses using our functional synthesis procedure π. Without the oracle, we would need to enumerate all partitions of clauses, as well as all partitions of attributes.

For our example document, block a is made of two clauses: $x = i$ and $i + z = y$. The oracle partitions R_a into subsets $s_0 \stackrel{\text{def}}{=} \{x = i\}$ and $s_1 \stackrel{\text{def}}{=} \{i + z = y\}$. Then the oracle partitions V_a into an input set $I_a \stackrel{\text{def}}{=} \{i, y\}$ and an output set $O_a \stackrel{\text{def}}{=} \{x, z\}$. Given these two partitions, we generate local functions for each set of clauses s_0 and s_1 using π. Of course, such functions are not guaranteed to exist. In this case, $\pi_{I_a,O_a}(s_0)$ yields the function $f_1 \stackrel{\text{def}}{=} x := i$, and $\pi_{I_a,O_a}(s_1)$ produces $f_2 \stackrel{\text{def}}{=} z := y - i$

We apply the same process on block b. Since R_b is made of a single clause, the oracle trivially partitions R_b into R_b itself. The oracle splits V_b into $I_b \stackrel{\text{def}}{=} \{x\}$ and $O_b \stackrel{\text{def}}{=} \{y\}$, then by applying $\pi_{I_b,O_b}(R_b)$, we obtain the function $f_3 \stackrel{\text{def}}{=} y := x$.

Recomposition (Step 3) The third step consists of constructing a global function functionalizing $rel(d)$ by selecting a subset of local functions and composing them together. This is the key step of GM synthesis.

Since the oracle produced exactly the necessary functions, we now merely need to order them to satisfy their dependencies. That is, for each local function, the attributes read must be computed before the function is applied. We encode function dependencies using a hypergraph whose vertices are attributes and whose edges represent local functions (Figure 3). 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 functions. Here, by applying f_1 first, then f_3, and finally f_2, we obtain the desired global function.

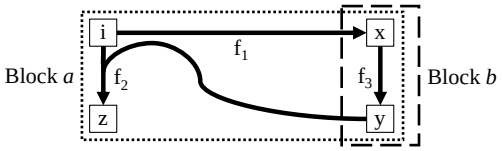


Figure 3. The hypergraph of the dependencies of f_1, f_2, and f_3. Note that the local function f_2 is represented by a hyperedge with two sources: i and y.

Implementing the Oracle Let's take a step back and analyze the role of the oracle. We relied on the oracle twice during the local synthesis step: the first time to partition block relations into subsets of clauses, and the second time to partition the attributes of each block into input/output sets. Each of these local oracular decisions must be coordinated to achieve global properties not apparent at the local (*i.e.*, block) level:

- *Function Selection* When looking at a block in isolation, we do not know how many local functions are needed to compute all of its attributes. In our example, the attributes of block a are computed with two local functions, 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 subsets of clauses, we would be restricting ourselves to solving block a with a single local function, which is not possible in our example.
- *Flow of Computation* While we know the overall (document) inputs, 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 functions. In fact, the same node may be traversed multiple times by the global function, each time invoking one local function, like the node (labeled) a in our example.

We used the oracle to simplify our synthesis algorithm which *conceptually* relies on global reasoning to synthesize local functions. To gain scalability, we restrict the generation of local functions to block-local reasoning. In the absence of a benevolent oracle, we synthesize local functions considering both all partitions of clauses into subsets and all partitions of attributes into input/output sets. As a result of this exhaustive enumeration, we obtain many more local functions than needed for the construction of the global function. We "implement" the oracle in the recomposition step, in which we must now select which local functions to use. We perform the selection symbolically, by reasoning on a hypergraph summarizing all flows of computation. By selecting local functions, we are indirectly making the same two decisions the oracle made: for each block, we select a clause partition and an input/output partition.

3.3 Modular Synthesis

We formalize the three steps of GM-synthesis (decomposition, local synthesis, and recomposition) for a language of a single document (Figure 4). Let d be this document.

Decomposition (Step 1) Conveniently, the structure of the document provides us with an initial decomposition where related constraints are already clustered together by the programmer: we decompose $rel(d)$ at nodes/blocks boundaries.

Note that 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 conjuncted with other relations to be functional. We discuss completeness of GM synthesis in Section 3.5.

Local Synthesis (Step 2) To start, let us define local functions formally.

Definition 4 (Local Function)**.** *Given a block* $(V, R \stackrel{\text{def}}{=} cl_0 \wedge \ldots \wedge cl_n)$*, a* local function *is a quadruple* (f, I, O, S) *where*

1. *I and O are lists of input/output attributes such that $I \subseteq V$, $O \subseteq V$, and $I \cap O = \varnothing$,*
2. *$S \subseteq \{cl_0, \ldots, cl_n\}$ is a subset of clauses,*
3. *f functionalizes S with respect to inputs I: $f = \pi_{I,O}(S)$.*

Note that executing the local function (f, I, O, S) assigns the attributes computed by f with values satisfying all clauses in S.

To generate as many local functions as possible, for each block (V, R) in d, we enumerate both all partitions of clauses of R and all input/output partitions of V, as detailed in Algorithm 1.

Algorithm 1: Synthesize local functions for a block.

Input: A block $b \stackrel{\text{def}}{=} (V, cl_0 \wedge \ldots \wedge cl_n)$
Output: A set of local functions over attributes V
begin
  $R \leftarrow \varnothing$
  foreach *subset* $S \subseteq \{cl_0, \ldots, cl_n\}$ **do**
    foreach *partition of* V *into sets* I *and* O **do**
      if $(f, I, O, S) = \pi_{I,O}(S)$ *exists* **then**
        Add (f, I, O, S) to R.
    end
  end
  return R
end

Recomposition (Step 3) We reduce the problem of choosing and composing local functions 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 5 (Hypergraph Summary)**.** *Given a document d, an hypergraph summary $H_d \stackrel{def}{=} (V, E)$ is such that V is the set of attributes of d and E is a set of local functions. Each local function (f, I, O, S) is represented with the hyperedge (I, O), where I is the set of source attributes and O the set of destination attributes.*

Since connections are equality constraints between sets of attributes, we can also represent them with local functions. Recall that, for now, each connection (A, B) is such that A and B are singleton. Let $A \stackrel{\text{def}}{=} \{a\}$ and $B \stackrel{\text{def}}{=} \{b\}$. The connection (A, B) is equivalent to $(id, A, B, \{a = b\})$ where id is the identity function.

We construct the hypergraph H_d as follows: For each node n in d labeled with block b, we instantiate the set of local functions of b on the attributes of n. Finally, we add two hyperedges per connection, one for each possible flow of values, either up or down in the document tree. Algorithm 2 details this process.

Before we define the d-solver in terms of paths in H_d, let us note the following two facts about the hypergraph summary

Algorithm 2: Construct a hypergraph summary encoding all possible compositions of local functions.

Input: A document d and a set of connections C
Output: A hypergraph summary of d
begin
  $E \leftarrow \varnothing$
  foreach *node* n *in* d *labeled with block* b **do**
    Add $\{(I, O) \mid (f, I, O, S) \in \text{Algo1}(b)\}$ to E.
  end
  foreach *connection* (A, B) *in* C **do**
    Add $\{(A, B), (B, A)\}$ to E.
  end
  return $(I_d \cup O_d, E)$
end

H_d. First, each hyperpath encodes a function reading its source attributes and computing its destination attributes.

Lemma 1. *Each hyperpath $p = f_0, \ldots, f_n$ in H_d encodes a function $f_p[I_p, O_p] = f_0 \circ \ldots \circ f_n$. Let I_i, O_i be the input/output sets of f_i, the ith function in p. Then $O_p = \bigcup_{0 \leq i \leq n} O_i$ and $I_p = (\bigcup_{0 \leq i \leq n} I_i) \setminus O_p$. From properties of hyperpaths, it follows that:*

1. *The dependencies of each local function on the path are satisfied. For each function 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 functions f_i and f_j in p such that $i \neq j$, we have $O_i \cap O_j = \varnothing$.*

Lemma 2. *Every function f_p defined by a hyperpath p in H_d satisfies the conjunction of clauses of its local functions. Let $p = f_0 \ldots f_n$ be a hyperpath representing function f_p. Let (f_i, I_i, O_i, S_i) be the ith function in p. Then f_p functionalizes $\bigwedge_{0 \leq i \leq n} S_i$. We say that f_p satisfies all clauses* traversed*.*

Lemma 2 follows directly from the fact that, by construction, each local function (f_i, I_i, O_i, S_i) functionalizes S_i. Finally, let us define the subset of paths which can be executed.

Definition 6 (Executable Path)**.** *A hyperpath p in H_d is* executable *iff it starts from the document inputs. That is, the function $f_p[I_p, O_p]$ encoded by p is such that $I_p \subseteq I_d$.*

We are now ready to state under which conditions a hyperpath encodes a d-solver. That is, a global function which functionalizes $rel(d)$ with respect to the document inputs I_d.

Definition 7. *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 p in H_d encodes a global function which functionalizes $rel(d)$ with respect to document input I_d.*

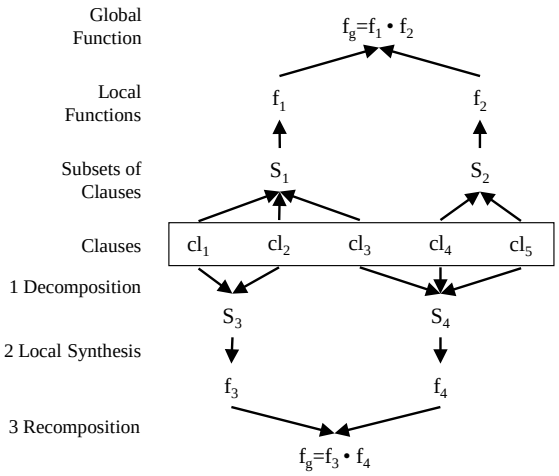


Figure 4. The three steps of GM synthesis. This diagram shows that two distinct decompositions can lead to syntactically different, yet semantically equivalent, d-solvers.

Since p is an executable spanning tree, it follows that both $I_p \subseteq I_d$ and $O_p = O_d$. From the coverage condition and using Lemma 2, we conclude that f_p functionalizes $rel(d)$.

Theorem 2. *If there exists an executable covering spanning tree in H_d, then $rel(d)$ is functional in I_d.*

Since every local function composing the covering spanning tree stems from a functional set of clauses (with respect to local function inputs), one can show that the set of all traversed clauses is functional with respect to I_d. Note that there may be multiple covering spanning trees. Each such tree encodes a semantically equivalent global function, but they may differ syntactically (Figure 4).

Together, Theorems 1 and 2 show that our approach is correct: the d-solvers synthesized always fulfil the specification. Note that finding a spanning tree in a hypergraph is NP-complete [29]. In the next subsection, we explain how to encode the search for a d-solver in SMT after generalizing our approach to languages of documents.

3.4 Grammar-Modular Synthesis

In this section, we generalize the modular synthesis technique presented so far to grammar-modular synthesis for languages of documents. In essence, to support grammars producing more than a single document, we need to handle alternative and recursive productions. By alternatives, we refer to non-terminals having more than one production. We start by formally defining languages of documents and language solvers.

Definition 8 (Language)**.** *A* language of documents *is regular tree-grammar $\mathcal{L}$ whose terminals are block-labeled nodes. Each tree in $\mathcal{L}$ forms a document. Each production of $\mathcal{L}$ can place semantic connections between attributes of a parent node and its children.*

Definition 9 ($\mathcal{L}$-Solver)**.** *Given a language $\mathcal{L}$, a $\mathcal{L}$-solver is a statically schedulable functional attribute grammar which defines a d-solver for every document $d \in \mathcal{L}$.*

A language of documents together with blocks definitions form a relational attribute grammar. As a result, we can view the synthesis of a $\mathcal{L}$-solver as converting a relational attribute grammar into a statically schedulable functional attribute grammar. As such, to construct a $\mathcal{L}$-solver, we compute: (i) the mode of all attributes together with a corresponding subset of local functions; and (ii) a total order over attributes. The modes capture whether attributes are inherited or synthesized. The total order prevents cyclic dependencies, which guarantees that the resulting functional attribute grammar is statically schedulable.

Synthetizing $\mathcal{L}$-Solvers Given a language $\mathcal{L}$, we synthesize a $\mathcal{L}$-solver as follows: First, we create a *witness* document which exhibits all productions of the grammar of $\mathcal{L}$. Then we create a hypergraph summary of $\mathcal{L}$ by applying Algorithm 2 on the witness document. Finally, from the hypergraph summary, we construct an SMT formula whose models encode both attribute modes and a subset of local functions. Together, they form a $\mathcal{L}$-solver.

The witness document can be produced easily by unrolling the grammar until every terminal (*i.e.*, node) appears in the document. By doing so, we ensure that $rel(d_w)$ contains all constraints of $\mathcal{L}$.

Connections across alternative productions can be encoded directly as hyperedges with multiple sources or destinations. Conveniently, properties of hyperpaths guarantee that all productions of the same non-terminal will share the same mode (*i.e.*, the same flow of computation). For example, consider the following language where block a may have either block b_1 or b_2 as child. Attribute $a.x$ is connected to either b_1.x or b_2.x.

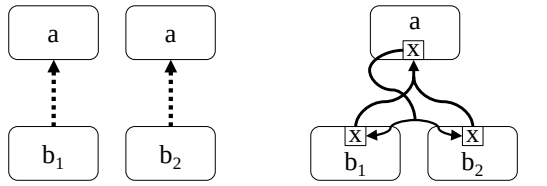
```
S ::= a(B) with a.x = B.x
B ::= b1() | b2()
```

We encode the two alternative productions of the non-terminal B with a single connection c: $(\{a.x\}, \{b_1.x, b_2.x\})_c$. When creating the hypergraph summary, Algorithm 2 encodes c with two hyper-edges (one with two destinations and one with two sources) representing values flowing either up or down through both derivations (Figure 5).

To handle recursion, we relax the definition of covering spanning trees (Definition 7) to carefully allow some cycles, those which are created by recursion and do not represent true cyclic dependencies of attributes.

SMT Encoding We encode the existence of a $\mathcal{L}$-solver as an SMT query.

Let d_w be the witness document of $\mathcal{L}$, and let $H_{d_w} = (V, E)$ be its hypergraph summary. Recall that V is the set of all attributes of d_w. Let $F \subseteq E$ be the set of local functions which are not connections, augmented with one function modeling inputs: $(-, \varnothing, I_{d_w}, \varnothing)$. Each local function



(a) The two derivable documents (b) Their hypergraph summary

Figure 5. A language of two documents, each stemming from one production of an alternative non-terminal (a), Algorithm 2 encodes the connection $(\{a.x\}, \{b_1.x, b_2.x\})$ with two hyperedges, thereby enforcing the same flow of computation for both documents (b).

$(f, I, O, S) \in F$ is encoded with one boolean flag e_f, which is true if f is used in the $\mathcal{L}$-solver; we say that f is *selected*.

We encode each attribute $x \in V$ with two variables:

1. One boolean m_x, representing the mode of x: m_x can either be inherited ($\downarrow$) or synthesized ($\uparrow$).
2. One integer l_x used to impose a total order on all attributes.

We partition the connections of $\mathcal{L}$ two subsets: (i) R, the set of recursive connections, those which stem from recursive nonterminals; and (ii) N, the set of non-recursive connections. Every connection $(A, B)_c \in N \cup R$ is encoded with one boolean m_c representing the mode of the connection: either inherited ($\downarrow$) or synthesized ($\uparrow$).

For each block (V, R) of $\mathcal{L}$, we encode each clause cl of R with one boolean named e_{cl}.

Finally, we define $bmode(x)$, a function converting the grammar mode of attribute x (inherited or synthesized) to a "block" mode (in or out) representing whether x is an input or an output of its block. The block mode is equivalent to modes of logic programs: attributes marked in are computed outside the block and propagated to it through connections; attributes marked out are computed within the block by a local function.

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

We break our encoding in five parts: (i) connections; (ii) local functions; (iii) the spanning property; (iv) schedulability; and (v) soundness. We explain each of them individually.

Connections The first part encodes the relationship between the mode of a connection and the mode of the attributes connected.

$$\phi_{\text{Conn}}((A,B)_c) := \left(m_c = \downarrow \implies \bigwedge_{x \in B} m_x = \downarrow\right) \ \wedge \ \left(m_c = \uparrow \implies \bigwedge_{x \in A} m_x = \uparrow\right)$$

Functions The second part is divided into two conjuncts: The first conjunct captures the relationship between local functions and attribute modes. Notice that we do not constrain the input of local functions to have an in mode. Doing so would prevent chaining of local functions within the same block, preventing the $\mathcal{L}$-solver from invoking multiple local functions during the same traversal. The second conjunct records all clauses of $rel(d_w)$ traversed by the subset of local functions selected.

$$\phi_{\text{Fun}}(f, I, O, S) := e_f \implies \bigwedge_{x \in O} bmode(x) = out \ \wedge \ \bigwedge_{cl \in S} e_{cl}$$

Spanning The third part guarantees that each attribute x is computed by a local function, a non-recursive connection, or inductively by recursion. Note that requiring every attribute to be computed at least once is not sufficient to ensure the soundness of the $\mathcal{L}$-solver. Consider the following grammar with two blocks $a \stackrel{\text{def}}{=} (\{x\}, x = 2)$ and $b \stackrel{\text{def}}{=} (\{x\}, x = 1)$:

```
S ::= a(B) with a.x = B.x
B ::= b()
```

Note the connection between the attributes $a.x$ and $b.x$. The only document derivable from this grammar has no solution. However, if we allowed attributes to be computed twice, then we would find a $\mathcal{L}$-solver which first assigns 1 to $b.x$ and then assigns 2 to $b.x$. This example illustrates how the same attribute may be assigned two distinct values, each satisfying one half of the specification. To reject such grammars, we require every attribute to be computed exactly once. As a result, our $\mathcal{L}$-solvers are *single-assignment* attribute grammars, a class of attribute grammars simpler to schedule. We define the logical connective $\odot$ to be true iff exactly one of its clauses is true.

$$\phi_{\text{Span}}(x) := \odot \left(\begin{array}{l} \{e_f \mid (f, I, O, S) \in F \wedge x \in O\} \cup \\ \{m_c = \downarrow \mid (A,B)_c \in N \wedge \ x \in B\} \cup \\ \{m_c = \uparrow \mid (A,B)_c \in N \wedge \ x \in A\} \cup \\ \{m_c = \uparrow \mid (A,B)_c \in R \wedge \ x \in B\} \end{array} \right)$$

Schedulability The fourth part guarantees the absence of cyclic dependencies by enforcing a total order on attributes. Note, that we only consider the non-recursive connections (N) to allow cycles of attributes caused by grammar recursion. That is, every cyclic path in subgraph of selected local functions must include one recursive connection.

$$\phi_{\text{Sched}} := \bigwedge_{(A,B)_c \in N} \left(m_c = \downarrow \implies \bigwedge_{x \in B} l_x > \max_{y \in A}(l_y)\right) \ \wedge \bigwedge_{(A,B)_c \in N} \left(m_c = \uparrow \implies \bigwedge_{x \in A} l_x > \max_{y \in B}(l_y)\right) \ \wedge \bigwedge_{(f,I,O,S) \in F} \left(I \supset \varnothing \wedge e_f \implies \bigwedge_{x \in O} l_x > \max_{y \in I}(l_y)\right)$$

Soundness The final part guarantees that the $\mathcal{L}$-solver functionalizes $rel(d_w)$. We ensure that the local functions selected

traverse (cover) all the clauses of (all the blocks of) $rel(d_w)$.

$$\phi_{\text{Sound}} := \bigwedge_{cl \in rel(d_w)} e_{cl}$$

$\mathcal{L}$-Solver Finally, by taking the conjunction of all five parts, we obtain a formula whose models encode both the subset of selected local functions (e_f variables) as well as modes for all attributes (m_x variables) and for all connections (m_c variables).

$$\phi := \bigwedge_{(A,B)_c \in N \cup R} \phi_{\text{Conn}}((A,B)_c) \ \wedge \ \phi_{\text{Sched}} \ \wedge \ \phi_{\text{Sound}} \ \wedge \bigwedge_{(f,I,O,S) \in F} \phi_{\text{Fun}}(f,I,O,S) \ \wedge \ \bigwedge_{x \in V} \phi_{\text{Span}}(x)$$

The translation of models of ϕ to functional attribute grammars is straightforward: The e_f booleans indicate which local functions to use. Note that ϕ also contains a static schedule of the attribute grammar encoded in the l_x variables. In general, our formalism is too abstract to model important execution characteristics like cache locality or parallelization opportunities. We throw away the schedule found and delegate this task to a dedicated attribute grammar scheduler [19].

3.5 Completeness

GM synthesis is correct (Theorems 1 and 2); the solvers generated are sound: they always satisfy the specification. However, GM synthesis is also incomplete and might fail to find a solver, even when one exists. In Section 4, we show that GM synthesis is sufficiently complete in practice.

Recall that GM synthesis relies on the following hypothesis: the global function is expressible as compositions of local functions. The granularity of the decomposition affects whether our hypothesis holds. Coarser initial decompositions (*i.e.*, blocks) yield more local functions at the expense of creating larger local synthesis problems, thus decreasing efficiency. Note that the number of local functions synthesized grows monotonically with the size of blocks only because we consider all subsets of clauses when performing local synthesis.

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. In the next two paragraphs, we state a condition for hierarchical linear systems of equations; this condition is necessary and sufficient to guarantee zero cost of modularity. Finally, we define a class of constraints for which modularity always incurs no cost. For clarity, we state these two properties considering a single document; they are generalizable by induction on the document grammar.

Linear Equations Without loss of generality, we abstract away connections: blocks share connected variables directly, as in the example of Section 3.2. We also assume that the local synthesis procedure π is complete.

Since we are limiting ourselves to linear equations, 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 3 (Completeness Condition)**.** *GM synthesis is complete for linear equations iff M_d can be triangularized (i) using row combinations (*i.e.*, adding a linear combination of rows to another) only between rows belonging to the same block and (ii) using row interchanges for any pair of rows.*

We give an outline of the proof. The first step is to show that the recomposition step of GM synthesis performs the equivalent of back-substitutions on M_d (assuming M_d is upper triangular). For linear equations, local synthesis reduces to row combinations within each block. Indeed, row combinations together with row interchanges form a complete quantifier elimination procedure for linear equations: Gaussian elimination. As such, the power of the local synthesis (π) is exactly row combinations. By requiring M_d to be triangular modulo row interchange after local synthesis, we ensure that $rel(d)$ is solvable with back-substitutions only.

Equality Constraints There exists a (very restricted) class of constraints for which modularity has no cost, regardless of the decomposition: equality constraints. If all atoms of $rel(d)$ are equalities between pairs of attributes, GM synthesis reduces to computing the equivalence classes of $rel(d)$. It is possible to show that equivalences classes are indirectly computed as part of the recomposition step, thus guaranteeing the completeness of modular synthesis.

Of course, equality constraints are too restrictive for all but the most trivial specifications. However, using the same line of reasoning, this result can be extended to demonstrate that the cost of modularity is not affected by the introduction of new equality constraints. As such, simple factorizations of the specification, such as breaking down large constraints into smaller ones have no effect on completeness; a reassuring property for specification authors.

4. Evaluation

In this section, we evaluate GM Synthesis along the following three axes:

- *Scalability and Completeness* Since GM synthesis trades completeness for scalability (to a degree controllable with the granularity of decomposition, see Section 3.5), is GM synthesis both scalable and complete enough to synthesize $\mathcal{L}$-solvers for realistic layout languages?
- *Performance* How does the solving speed of our $\mathcal{L}$-solvers compare with state-of-the-art, general-purpose constraint solvers? How do $\mathcal{L}$-solvers and general-purpose constraint solvers scale as document size increases?
- *Parameterizable Layout Engines* Can our layout specifications be reused for multiple $\mathcal{L}$-solvers, each synthesized for a different set of input attributes, one per user interaction (*e.g.*, resize)? This benefit results from using non-directional constraints which capture flows of values in several directions.

Experimental Setup GM synthesis is parametrized by the local synthesis procedure π. In our experiments, we implemented π with a combination of Sketch [26] for linear relations and Gröbner Bases (from Mathematica) for polynomial equations. There are many other procedures which could

be used to implement π. We note Comfusy [15] and Mjollnir [20].

We used the Superconductor attribute grammar scheduler [19] 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 1(b) and 1(c) have been laid out by one of our $\mathcal{L}$-solvers.

All our benchmarks were run on a 2.5GHz Intel Sandy Bridge processor with 8Gb of RAM.

Scalability and Completeness To show that GM synthesis is widely applicable, we demonstrate it on layout languages drawn from the three major layout domains. Our case studies cover: (i) document (webpage) layout; (ii) Graphical User Interface (GUI); and (iii) data visualization. Each of the three languages presented below is full-fledged and computes all attributes needed for rendering.

1. Our first case study is a guillotine layout language where a set of horizontal and vertical dividers partition the space. A subset of CSS can be encoded in such languages [25]. The guillotine language totals 30 constraints. This is the only language in which all constraints are linear.
2. Our second case study is a language of flexible grids [9]. Such languages are frequently used to layout widgets in graphical user interfaces [12]. The sizes of each cell of the grid are allocated based on a weighted sum, producing non-linear constraints. The weight of each cell is a runtime input. The grid language consists of 47 constraints.
3. Finally, a language of treemaps [13], 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.

Our GM synthesizer is sufficiently complete to successfully generate a $\mathcal{L}$-solver for each of the three case studies. The synthesis took less than five minutes, an acceptable compilation time, with the local synthesis step and the recomposition step using approximately equal halves. To illustrate the complexity of the $\mathcal{L}$-solvers obtained after scheduling, Table 1 lists the number of tree traversals, the number of local functions used, and size of the JavaScript code. For reference, Firefox's layout engine for CSS uses four passes [3]. Finally, the number of lines of code reported includes only the layout engine itself (*i.e.*, the computation of document attributes); code related to rendering has been explicitly excluded.

We also compare our work with direct functional synthesis techniques, such as Comfusy and Sketch. Such techniques are limited to synthesis of d-solvers, they do not generalize to languages of documents. As such, we apply them on a single small document of 127 nodes. Neither Comfusy nor Sketch could synthesize a d-solver in less than one hour. These results indicate that GM synthesis strikes the right balance between completeness and scalability of synthesis for our domain.

Language	**Tree Traversals**	**Local Functions**		**SLOC**
		Total	Selected	
Guillotine	td	289	74	189
Grid	td ; bu ; td	385	89	283
Treemap	td ; bu ; td ; bu ; td	394	91	341

Table 1. The complexity of $\mathcal{L}$-solvers for each of our three case studies. The second column shows the number and type of tree passes over the document: td denotes a top-down pass and bu a bottom-up one. The third columns reports the number of local functions synthesized and the number of local functions used. Finally, the fourth column shows the number of lines of JavaScript code.

Performance We compare the performance of our synthesized $\mathcal{L}$-solvers with Z3 [21], a state-of-the-art constraint solver. Note that our solvers are implemented in JavaScript, a relatively slow language. Z3 solves the constraint system defined by the document ($rel(d)$) at runtime. In essence, we measure the ability of GM synthesis to shift the cost of solving constraints from runtime to compile time.

We measured the time to compute the layout of documents from 255 to 16383 nodes, for each of the 3 layout languages outlined above. We argue that such document sizes are typical: the front page of `www.nytimes.com` contains over 3000 nodes and data-visualizations tend to be much larger. For each case study, we chose the fastest SMT theory which could express the layout specification. Interestingly, the non-linear arithmetic solver was faster than bivectors 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 milliseconds for typical document sizes. Missing entries (—) indicate "unknown" answers (Z3 produced no model). Notice that our $\mathcal{L}$-solvers scale linearly with the document size.

Our $\mathcal{L}$-solvers scale linearly with size of the document, whereas Z3 exhibits exponential behavior on the largest (16383 nodes) document for all three languages. This asymptotic 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. On the largest document, Z3 was unable to compute a layout within 3 minutes (either timing out or reporting "unknown") for all three case studies. Our results show that across the three

case studies, our $\mathcal{L}$-solvers are fast enough (<0.5 second) for interactive settings.

Parameterizable Layout Engines We illustrate the expressiveness of non-directional constraints by synthesizing multiple $\mathcal{L}$-solvers from the same specification. Each solver responds to a distinct event or user-interaction by updating the layout. For instance, when the user resizes the main window, one $\mathcal{L}$-solver recomputes the layout using the new width and height as input. We illustrate the power of non-directionality on our language of treemaps.

Imagine a treemap representing the market capitalization of companies. The leaves of the document are companies while inner nodes encode the tiling of the screen (Figure 1(a)). Let's consider the following two events: (i) the values of all companies are updated; and (ii) the user resizes the treemap.

Each event defines its own set of runtime inputs from which all remaining attributes are computed. 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 the same specification, our synthesizer generates two $\mathcal{L}$-solvers, one per set of runtime inputs. For the first event, we obtain (after scheduling) a five pass $\mathcal{L}$-solver, whereas the second event yields a three pass $\mathcal{L}$-solver.

The ability to capture multiple flows of computation within the same specification indicates that relational attribute grammars are a concise formalism for expressing interactive layouts.

5. 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 [17, 18], is an instance of the AE-paradigm, also known as the Skolem paradigm for synthesis [23]. GM synthesis builds upon functional synthesis procedures, such as Comfusy [15] or Sketch [26], by enabling modular decompositions of specifications to gain scalability.

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 [24] and QuickPlan [28] have been proposed. In CP, each "planning constraint" corresponds to a set of clauses in our framework. Similar to our d-solver setting, given a set of planning constraints, each associated with local functions (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 functions as well as partitioning relations, to satisfy special requirements of the algorithm. QuickPlan works in quadratic-time by imposing a clever restriction on planning constraints: each local function must mention all variables of its planning constraint, either as input or as output. The programmer satisfies this restriction by intelligently factoring clauses into planning constraints when writing local functions. In our setting, the same information is left to the oracle (*i.e.*, we search over the space of all factorizations). As illustrated in Section 3.2, our oracle partitions the relation of each block into subsets of clauses, each corresponding to one planning constraint. Without this step, we would be restricted to computing all attributes of each block with a single local function, which would prevent creating layout engines 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 functions for all subsets of clauses. As a result, we obtain many more local functions than in the traditional constraint planning setting. Naively encapsulating local functions into planning constraints meeting QuickPlan's simplifying assumption would create an exponential explosion. With one planning constraint per subset of clauses, QuickPlan's complexity would become $(2^n)^2$ where n is the number of clauses. In general, constraint planning for non-directional constraints is NP-complete [16].

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 attribute grammars and logic programming. Deransart *et al.* [8] give theoretic constructions demonstrating how relational grammars, functional grammars and directed clause programs are related to one another. Mode analysis [7] techniques for logic programs, which compute whether clause arguments of logical programs are input or output, could be — in principle — transposed to attribute grammars 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 functions, which can be viewed as sets of possible modes for a block. Local functions are computed independently for each block and can be reused across layout languages. Mode analysis techniques based on abstract interpretation operate on the whole program.

Constraint Logic Programming (CLP) In constraint logic programming ([1, 2, 30]), 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 and a valuation of its inputs, CLP tools search for one layout (*i.e.*, solution) among the potentially many, whereas we ensure that the specification is functional with respect to document inputs. That is, the layout is uniquely determined by inputs (*i.e.*, deterministic).

6. 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 applied GM synthesis to document layout and generated 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. In our experiments, the theoretical incompleteness of GM synthesis did not materialize. 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 an attribute grammar-based type-checker from relational type system specifications. The underlying domain of attributes would have to be extended to data types richer than Rationals.

References

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

[2] K. R. Apt and M. Wallace. *Constraint Logic Programming using Eclipse*. Cambridge University Press, New York, NY, USA, 2007.

[3] E. Atkinson. personal communication, 2014.

[4] G. J. Badros, A. Borning, K. Marriott, and P. Stuckey. Constraint cascading style sheets for the web. UIST, pages 73–82, 1999.

[5] M. Bostock, V. Ogievetsky, and J. Heer. D3 data-driven documents. *IEEE Trans. on Visualization and Computer Graphics*, pages 2301–2309, 2011.

[6] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007.

[7] S. K. Debray and D. S. Warren. Automatic mode inference for logic programs. *The Journal of Logic Programming*, pages 207 – 229, 1988.

[8] P. Deransart and J. Małuszynski. Relating logic programs and attribute grammars. *The Journal of Logic Programming*, pages 119 – 155, 1985.

[9] S. K. Feiner. A grid-based approach to automating display layout. In *Proceedings on Graphics Interface '88*, pages 192–197, 1988.

[10] J. Heer and M. Bostock. Declarative language design for interactive visualization. *InfoVis*, pages 1149–1156, 2010.

[11] T. Hottelier, R. Bodik, and K. Ryokai. Programming by manipulation for layout. UIST'14, 2014.

[12] N. Hurst, W. Li, and K. Marriott. Review of automatic document formatting. DocEng, pages 99–108, 2009.

[13] B. Johnson and B. Shneiderman. Tree-maps: a space-filling approach to the visualization of hierarchical information structures. In *Visualization*, 1991.

[14] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, pages 127–145, 1968.

[15] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Complete functional synthesis. In *PLDI '10*, pages 316–329, 2010.

[16] J. Maloney. *Using Constraints for User Interface Construction*. Department of Computer Science: Technical report. University of Washington, Department of Computer Science, 1991.

[17] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Trans. Program. Lang. Syst.*, pages 90–121, 1980.

[18] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, pages 151–165, 1971.

[19] L. A. Meyerovich, M. E. Torok, E. Atkinson, and R. Bodik. Parallel schedule synthesis for attribute grammars. PPoPP '13, pages 187–196, 2013.

[20] D. Monniaux. A Quantifier Elimination Algorithm for Linear Real Arithmetic. In *LPAR '08*, pages 243–257, 2008.

[21] L. D. Moura and N. Bjørner. Z3: An efficient smt solver. TACAS'08/ETAPS'08, pages 337–340, 2008.

[22] B. A. Myers, B. V. Zanden, and R. B. Dannenberg. Creating graphical interactive application objects by demonstration. UIST, pages 95–104, 1989.

[23] A. Pnueli and R. Rosner. On the synthesis of a reactive module. POPL '89, pages 179–190, 1989.

[24] M. Sannella. Skyblue: a multi-way local propagation constraint solver for user interface construction. UIST, pages 137–146, 1994.

[25] N. Sinha and R. Karim. Compiling mockups to flexible uis. ESEC/FSE, pages 312–322, 2013.

[26] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat. Combinatorial sketching for finite programs. In *ASPLOS '06*, pages 404–415, 2006.

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

[28] B. Vander Zanden. An incremental algorithm for satisfying hierarchies of multiway dataflow constraints. *ACM Trans. Program. Lang. Syst.*, pages 30–72, 1996.

[29] D. M. Warme. *Spanning Trees in Hypergraphs with Applications to Steiner Trees*. PhD thesis, 1998.

[30] R. H. C. Yap. Constraint processing by rina dechter. *Theory Pract. Log. Program.*, pages 755–757, 2004.

[31] C. Zeidler, C. Lutteroth, W. Sturzlinger, and G. Weber. The auckland layout editor: An improved gui layout specification process. UIST, pages 343–352, 2013.