Searching Entangled Program Spaces

JAMES KOPPEL, MIT, USA
ZHENG GUO, UC San Diego, USA
EDSKO DE VRIES, Well-Typed LLP, United Kingdom
ARMANDO SOLAR-LEZAMA, MIT, USA
NADIA POLIKARPOVA, UC San Diego, USA

Many problem domains, including program synthesis and rewrite-based optimization, require searching astronomically large spaces of programs. Existing approaches often rely on building specialized data structures—version-space algebras, finite tree automata, or e-graphs—to compactly represent such spaces. At their core, all these data structures exploit independence of subterms; as a result, they cannot efficiently represent more complex program spaces, where the choices of subterms are entangled.

We introduce *equality-constrained tree automata* (ECTAs), a new data structure, designed to compactly represent large spaces of programs with entangled subterms. We present efficient algorithms for extracting programs from ECTAs, implemented in a performant Haskell library, ECTA. Using the ECTA library, we construct HECTARE, a type-driven program synthesizer for Haskell. HECTARE significantly outperforms a state-of-the-art synthesizer HOOGLE+—providing an average speedup of 8×—despite its implementation being an order of magnitude smaller.

CCS Concepts: • **Theory of computation → Tree languages**; • **Software and its engineering → Functional languages**; **Automatic programming**.

Additional Key Words and Phrases: program synthesis, e-graphs, Haskell, type systems

ACM Reference Format:
James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces. *Proc. ACM Program. Lang.* 6, ICFP, Article 91 (August 2022), 29 pages. https://doi.org/10.1145/3547622

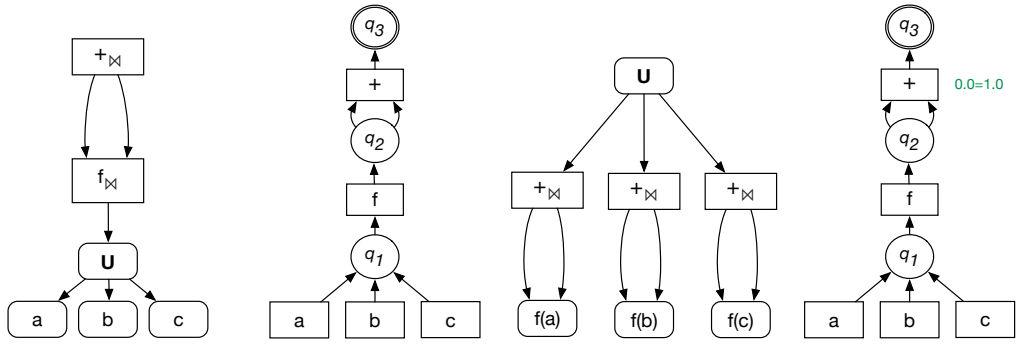
1 INTRODUCTION

From program synthesis to theorem proving and compiler optimizations, a range of problem domains make use of data structures that compactly represent large spaces of terms. In program synthesis, the most well-known example is *version space algebras* (VSAs) [Lau et al. 2003; Polozov and Gulwani 2015], the data structure behind the successful spreadsheet-by-example tool FLASHFILL [Gulwani 2011]. Although there may be over 10^{100} programs matching an input/output example, FLASHFILL is able to represent all of them as a compact VSA, efficiently run functions over every program in the space, and then extract the best concrete solution.

To illustrate the idea behind VSAs, consider the space of nine terms $\mathcal{T} = \{\mathsf{f}(t_1) + \mathsf{f}(t_2)\}$ where $t_1, t_2 \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$. Fig. 1a shows a VSA that represents this space. In a VSA a *union node*, marked with $\cup$, represents a union of all its children, while a *join node*, marked with $\bowtie$, applies a function

Authors' addresses: James Koppel, MIT, Cambridge, MA, USA, jkoppel@mit.edu; Zheng Guo, UC San Diego, La Jolla, CA, USA, zhg069@eng.ucsd.edu; Edsko de Vries, Well-Typed LLP, United Kingdom, edsko@well-typed.com; Armando Solar-Lezama, MIT, Cambridge, MA, USA, asolar@csail.mit.edu; Nadia Polikarpova, UC San Diego, La Jolla, CA, USA, npolikarpova@eng.ucsd.edu.

2475-1421/2022/8-ART91
https://doi.org/10.1145/3547622



(a) VSA for $\mathcal{T}$ (b) FTA for $\mathcal{T}$ (c) VSA for $\mathcal{U}$ (d) ECTA for $\mathcal{U}$

Fig. 1. Representations of $\mathcal{T} = \{\mathsf{f}(t_1) + \mathsf{f}(t_2)\}$ and $\mathcal{U} = \{\mathsf{f}(t) + \mathsf{f}(t)\}$, where $t, t_1, t_2 \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$.

symbol to every combination of terms represented by its children. You can see how, by exploiting the shared top-level structure of the terms in $\mathcal{T}$, this VSA is able to compactly represent nine terms, each of size five, using only six nodes.

Another data structure that exploits sharing in a similar way is *e-graphs*, which enjoy a wide range of applications, including theorem proving [Detlefs et al. 2005], rewrite-based optimization [Tate et al. 2009], domain-specific synthesis [Nandi et al. 2020, 2021], and semantic code search [Premtoon et al. 2020]. Both VSAs and e-graphs are now known [Koppel 2021; Pollock and Haan 2021] to be equivalent to special cases of *finite tree automata* (FTAs), which have independently experienced a surge of interest in recent years [Adams and Might 2017; Wang et al. 2017, 2018]. Fig. 1b shows an FTA that represents the same term space as the VSA in Fig. 1a. An FTA consists of *states* (circles) and *transitions* (rectangles), with each transition connecting zero or more states to a single state. Intuitively, FTA transitions correspond to VSA's join nodes, and FTA states correspond to VSA's union nodes (although in a VSA, union nodes with a single child are omitted). Importantly, all three data structures[1] thrive on spaces where terms share some top-level structure, while their divergent sub-terms can be chosen *independently* of each other.

Challenge: Dependent Joins. Consider now the term space $\mathcal{U} = \{\mathsf{f}(t) + \mathsf{f}(t)\}$, where $t \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$, that is, a sub-space of $\mathcal{T}$ where both arguments to f *must be the same term*. Such "entangled" term spaces arise naturally in many domains. For example, in term rewriting or logic programming, we might want to represent the subset of $\mathcal{T}$ that matches the non-linear pattern $X + X$. Similarly, in type-driven API search [Gissurarson 2018; Mitchell 2004], we might want to represent the space of all *types* of library functions that unify with a given query type, such as `List` $\alpha \to$ `List` α.

Existing data structures are incapable of fully exploiting shared structure in such entangled spaces. Fig. 1c shows a VSA representing $\mathcal{U}$: here, the node $+_{\bowtie}$ cannot be reused because VSA joins are *independent*, whereas our example requires a dependency between the two children of +. This limitation is well-known: for example, the seminal work on VSAs [Lau et al. 2003] notes that "efficient representation of non-independent joins remains an item for future work."

Solution: ECTA. To address this limitation, we propose a new data structure we dub *equality-constrained tree automata* (ECTAs). ECTAs are tree automata whose transitions can be annotated with *equality constraints*.[2] For example, Fig. 1d shows an ECTA that represents the term space $\mathcal{U}$. It

[1] We omit e-graphs from Fig. 1 for space reasons, but also because e-graphs are typically used to represent congruence relations rather than arbitrary sets of terms, which makes them less relevant to our setting, as we discuss in §9.

[2] This might remind some readers of Dauchet's *reduction automata*; we postpone a detailed comparison to related work (§9).

is identical to the FTA in Fig. 1b save for the constraint 0.0 = 1.0 on its + transition. This constraint restricts the set of terms accepted by the automaton to those where the sub-term at path 0.0 (the first child of the first child of +) equals the sub-term at path 1.0 (the first child of the second child of +). The constraint enables this ECTA to represent a dependent join while still fully exploiting shared structure, unlike the VSA in Fig. 1c.

Challenge: Enumeration. Being able to represent a term space is not particularly useful unless we also can efficiently *extract* a concrete inhabitant of this space—or, more generally, *enumerate* some number of its inhabitants. Unsurprisingly, equality constraints make enumeration harder, since the terms must now comply with those constraints (in fact, as we demonstrate in §7.1, extracting a term for an ECTA is at least as hard as SAT solving). A naïve fix is to filter out spurious (constraint-violating) terms after the fact, but such "rejection sampling" can be extremely inefficient.

Solution: Dynamic and Static Reduction. Our first insight for how to speed up enumeration is inspired by constraint-based type inference. Instead of making an *eager* choice at a constrained state, such as q_1 in Fig. 1d, our enumeration technique *postpones* this choice, instead introducing a "unification variable" V_1 to stand for the chosen term. This variable gets reused the second time q_1 is visited. At the end, V_1 is reified into a concrete term, thereby making a simultaneous choice at the two constrained states, which is guaranteed by construction to satisfy all equality constraints. We dub this mechanism *dynamic reduction*, where "dynamic" refers to operating during the enumeration process. As we illustrate in §2, dynamic reduction becomes more involved when equality constraints relate different states: in that case the term space associated with a unification variable gets refined during enumeration.

Our second insight is that enumeration can often be made even more efficient by transforming the ECTA *statically*—that is, before the enumeration starts—so that some of its constraints are "folded" into the structure of the underlying FTA. We will present examples in §2 of using static reduction to "prune" away states that cannot be part of any term that satisfies the constraints.

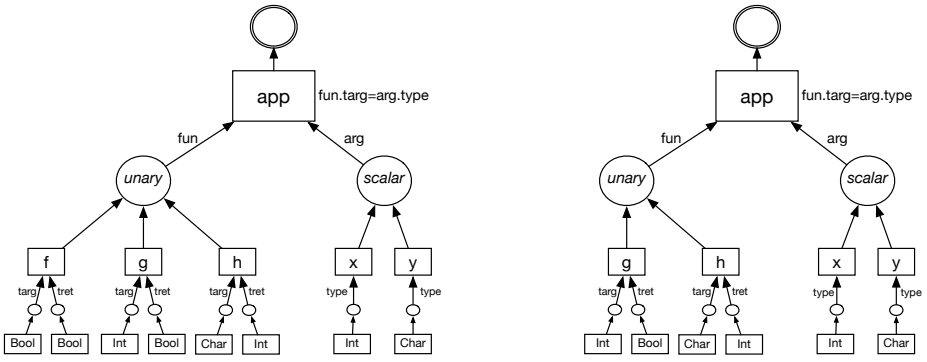
Contributions. In summary, this paper makes the following contributions:

(1) We introduce the *ECTA data structure* (§3), which supports compact representation of program spaces with dependent joins, as well as efficient enumeration (§4) via *static* and *dynamic reduction*. We first formalize the simpler acyclic ECTAs, and then show how to add cycles in order to support infinite term spaces (§5).
(2) We develop *ECTA encodings* for two diverse domains: Boolean satisfiability and type-driven program synthesis (§7). These encodings illustrate that ECTAs are expressive and versatile, and that ECTA enumeration can effectively be used as a general-purpose constraint solver.
(3) We implement the data structure and its operations in a performant Haskell library, ECTA.

We evaluate the ECTA library on the domain of type-driven program synthesis (§8). The experiments show that our ECTA-based synthesizer HECTARE significantly outperforms its state-of-the-art competitor HOOGLE+ [Guo et al. 2020], despite our implementation being *only a tenth of the size*. Specifically, HECTARE is able to solve 88% of synthesis problems in the combined benchmark suite compared to only 64% by HOOGLE+, and on commonly solved benchmarks HECTARE is 8× *faster* on average. Further, our evaluation demonstrates that static and dynamic reduction are critical for performance: ablating either of those mechanisms reduces the number of benchmarks solved, while a naïve baseline that uses "rejection sampling" enumeration is unable to solve *any* benchmarks.

2 ECTA BY EXAMPLE

In this section we illustrate the ECTA data structure and its two major features—static and dynamic reduction—using the problem of type-driven program synthesis as a motivating example. We give a simple encoding of the space of well-typed small programs into ECTAs, and then show how the



(a) Initial ECTA. (b) Reduced ECTA.

Fig. 2. ECTAs representing all well-typed size-two terms in the environment Γ_1.

general-purpose ECTA operations are used to efficiently enumerate the well-typed terms. We will present the full encoding, which also handles arbitrary prenex polymorphism and higher-order functions, in §7, along an with ECTA encoding of another problem domain.

2.1 Representing Spaces of Well-Typed Terms

Consider a typing environment $\Gamma_1 = \{x : \texttt{Int}, y : \texttt{Char}, f : \texttt{Bool} \to \texttt{Bool}, g : \texttt{Int} \to \texttt{Bool}, h : \texttt{Char} \to \texttt{Int}\}$. Suppose we are interested in enumerating all application terms that are well-typed in Γ_1; for now let us restrict our attention to terms of size two—that is, applications of variables to variables. The space of all such terms can be compactly represented with an ECTA, as shown in Fig. 2a.

This ECTA has a *transition* for each variable in Γ_1; scalar variables (x and y) are annotated with their `type`, while functions (f, g, and h) are annotated with an argument type `targ` and a return type `tret`. The *node* (state) *unary* represents the space of all unary variables, while the node *scalar* represents the space of all scalars. The accepting node has a single incoming transition `app`, which represents an application of a unary `fun` to a scalar `arg`, fulfilling the restriction to size-two terms.[3]

While the underlying tree automaton of this ECTA (its *skeleton*) accepts all terms of the form $A\ B$ where $A \in \{f, g, h\}$ and $B \in \{x, y\}$; most of these terms, such as $f\ x$ are ill-typed. In order to restrict the set of represented terms to only well-typed ones, there is an *equality constraint* `fun.targ = arg.type` attached to the `app` transition, which demands that the types of the formal and the actual arguments coincide. Thanks to this constraint, the full ECTA accepts only the two well-typed terms, $g\ x$ and $h\ y$. (Note that in this presentation, we give names to the incoming edges of each transition to make the constraints more readable; in the formalization, we instead use indices to refer to the edges.)

2.2 Static Reduction

How would one go about enumerating the terms represented by the ECTA in Fig. 2a? A naïve approach is to (1) enumerate all terms represented by its skeleton and (2) filter out those terms that violate the constraint. Step 1 is easily accomplished via depth-first search, starting from the root (the accepting node) and picking a single incoming transition for every node. This approach is, however, inefficient: it ends up constructing six terms, only to filter out four of them. In ECTA terminology, the skeleton admits six *runs*, four of which are *spurious* (violate the constraints).

[3]In our full encoding in §7 we remove the distinction between the terms of different arity in order to support higher-order and partial applications.

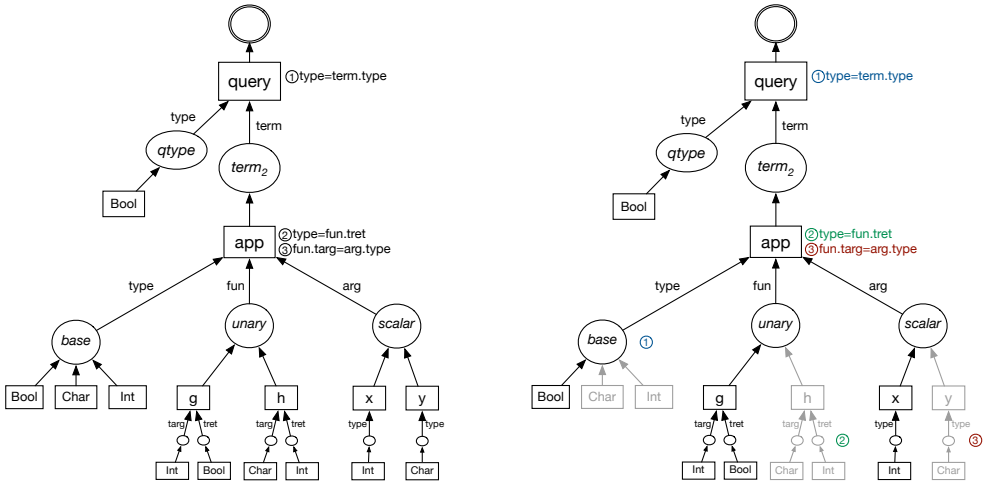


Fig. 3. ECTAs representing size-two terms of type `Bool`. The diagram on the right shows how a sequence of static reductions on the constraints ①, ②, and ③ eliminates the grayed-out transitions and nodes.

Our **first key insight** is that enumeration can often be made more efficient by transforming the ECTA's skeleton so as to reduce the number of spurious runs. We refer to this transformation as *static reduction* (because it happens once, *before* the enumeration starts). The reduced ECTA for our example is given in Fig. 2b. Intuitively, we were able to eliminate the f transition entirely because there are no scalar variables that match its formal argument type `Bool`; as a result the reduced ECTA contains only two spurious runs instead of four.

More formally, static reduction works via automata *intersection*. For the ECTA in Fig. 2a, reducing the constraint `fun.targ = arg.type` involves constructing an automaton that accepts all terms reachable via the path `arg.type`—namely `Int` and `Char`—and intersecting it with each node at the path `fun.targ`. Since the child node of `f` labeled `targ` represents only `Bool`, the intersection for that child is empty, meaning the `f` transition can never be used to satisfy the constraint, and hence can be eliminated. The reduction algorithm performs a similar intersection for `g` and `h`, as well as (in the other direction) `x` and `y`, but finds that each of these other choices could be part of a satisfying run, and eliminates no further transitions.

2.3 Type-Driven Program Synthesis with ECTAs

In type-driven program synthesis, we are typically not interested in all well-typed terms, but rather terms of a given *query* type. The ECTA in Fig. 3 (left) represents a type-driven synthesis problem with the same environment Γ_1 as before and query type `Bool`. The main difference between this automaton and the one in Fig. 2b is the new transition `query`, whose `type` edge encodes the given query type and whose `term` edge connects to the node representing all well-typed terms in the search space. To filter out the terms of undesired types, constraint ① prescribes that the `term`'s `type` be equal to the query type. In order for this constraint to make sense, we also add a `type` annotation to the `app` transition; the type of an application is initially undetermined (can be any base type), but is restricted by a new constraint ② to coincide with the return type of the function.

Fig. 3 (right) demonstrates a sequence of static reductions that happens to eliminate *all* spurious run of this ECTA, until its skeleton represents the sole solution to the synthesis problem: the term $g\ x$. First, reducing constraint ① eliminates all possible types of the application except `Bool`; next,

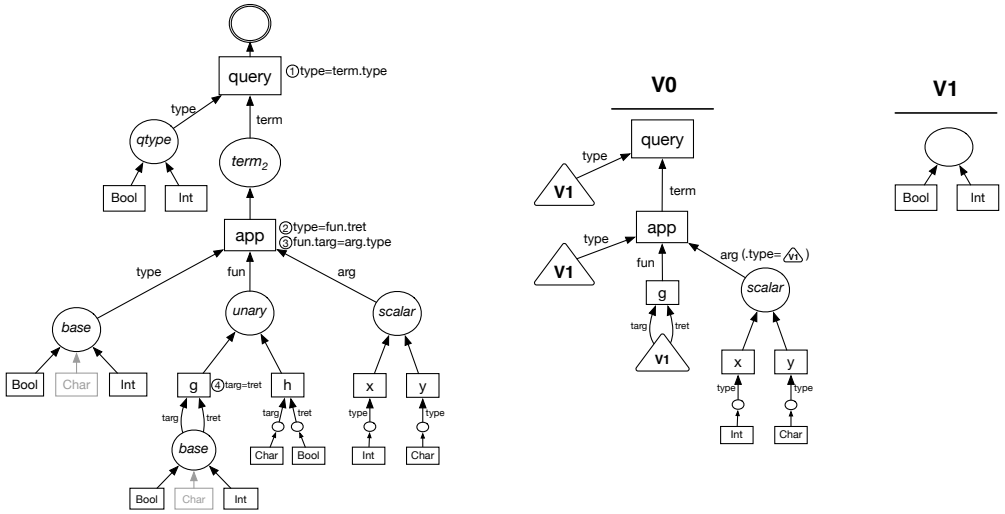


Fig. 4. Type-driven synthesis with polymorphic functions. Left: ECTA representing all terms of types `Bool` or `Int` in Γ_2. Grayed-out transitions are eliminated by static reduction. Right: Intermediate state during enumeration. The choice of the query type has been suspended into an auxiliary automaton V_1.

reducing ② eliminates the function h as it has a wrong return type; finally, reducing ③ eliminates the argument y, since it is incompatible with the only remaining function g.

2.4 Dynamic Reduction

In the previous example, static reduction was able to eliminate all spurious runs of the ECTA before enumeration. so that no spurious runs remained. This is not always possible. Consider a slightly more involved version of type-driven synthesis where functions can be polymorphic. Specifically let $\Gamma_2 = \{x : \texttt{Int}, y : \texttt{Char}, g : \forall\alpha.\alpha \to \alpha, h : \texttt{Char} \to \texttt{Bool}\}$, and suppose we are interested in all size-two terms of types `Int` or `Bool`. This problem can be represented by the ECTA in Fig. 4, which is similar to the one in Fig. 3. The only interesting difference is how the polymorphic type of g is represented: the type variable α is encoded as a *union* of all types it can unify with—here `Bool`, `Char`, and `Int`.[4] Crucially, the constraint ④ on g guarantees that *the same* type is used to instantiate both occurrences of α.

Although static reduction can eliminate some of the transitions in this ECTA (shown in gray), a fair number of spurious runs remain. For example, a naïve left-to-right enumeration would first pick `Bool` as the query type and g as the function (forcing the selection of $\texttt{Bool} \to \texttt{Bool}$ as the type of g), only to discover later that there is no argument of type `Bool`. More generally, in the presence of constraints, the choices made for constrained nodes are not independent, and making a wrong combination of choices early on (such as `Bool` and g in our example) may lead to expensive backtracking further down the line.

Our **second key insight** is that such backtracking can be avoided by *deferring* the enumeration of constrained nodes until more information is available. Fig. 4 (right) illustrates this idea. It depicts a *partially enumerated* term from the ECTA on the left. You can think of a partially enumerated term as a tree fragment at the top with yet-to-be-enumerated ECTAs among the branches. Importantly,

[4]Here we consider a limited form of polymorphism, where type variables can be instantiated only with base types; this restriction is relaxed in §7.

because the node *qtype* is constrained (by ①), it is not enumerated eagerly, but instead *suspended* into a named sub-automaton V_1. As the enumeration encounters each other node n constrained to be equal to *qtype* (via ①, ②, and ④), n is replaced by a reference to V_1, while V_1 is updated to $V_1 \sqcap n$. Thus, to arrive at Fig. 4 (right), the enumeration has made a single decision—picking g over h—whereas all the other choices have been deferred.

Finally, the enumeration picks x among the two *scalar*s. The `type` state of x—let's call it n_x—represent a singleton {`Int`} *and* is constrained to equal V_1 (by ③). As a result, V_1 gets intersected with n_x, eliminating its `Bool` alternative. Now when it comes time to "unsuspend" V_1, it only contains a single alternative, `Int`, which is already guaranteed to be consistent with all constraints. In other words, we have found the solution $g\ x$[5] without having to explicitly search over all possible query types, result types of the application, or instantiations of g; instead all these three choices were made simultaneously and consistency. We refer to this mechanism as *dynamic reduction* because it reduces the number of explored spurious runs *during* enumeration.

3 ACYCLIC ECTA

This section formalizes the ECTA data structure and its core algorithms. We begin by presenting the special case of ECTAs without cycles, which simplifies both the theory and implementation. Proofs of all theorems omitted from this and the following sections can be found in the extended version [Koppel et al. 2022].

3.1 Preliminaries

We first present standard definitions of terms, paths, and the prefix-free property from the term-rewriting literature.

Terms. A *signature* Σ is a set of function symbols, each associated with a natural number by the arity function. $\mathcal{T}(\Sigma)$ denotes the set of *terms* over Σ, defined as the smallest set containing all $s(t_0, \ldots, t_{k-1})$ where $s \in \Sigma$, $k = \text{arity}(s)$, and $t_0, \ldots, t_{k-1} \in \mathcal{T}(\Sigma)$. We abbreviate nullary terms of the form $s()$ as s.

Paths. Paths are used to denote locations inside terms. Formally, a *path* p is a list of natural numbers $i_1.i_2.\ldots.i_k \in \mathbb{N}^*$. The empty path is denoted ϵ, and $p_1.p_2$ denotes the concatenation of paths p_1 and p_2. We write $p_1 \sqsubseteq p_2$ if p_1 is a prefix of p_2 (and $p_1 \sqsubset p_2$ if it is a proper prefix). A set P of paths is *prefix-free* if there are no $p_1, p_2 \in P$ such that $p_1 \sqsubset p_2$.

Given a term $t \in \mathcal{T}(\Sigma)$, a *subterm of* t *at path* p, written $t|_p$, is inductively defined as follows: (i) $t|_\epsilon = t$ (ii) $s(t_0, \ldots, t_{k-1})|_{i.p} = t_i|_p$ if $i < k$ and $\bot$ otherwise. For example, for $t = +(f(a), f(b))$: $t|_{0.0} = a$, $t|_{1.0} = b$ and $t|_{2.0} = \bot$.

3.2 Path Constraints and Consistency

The difference between ECTAs and conventional tree automata is the presence of path equalities, such as $0.0 = 1.0$ in Fig. 1d. We now formalize the semantics of these path equalities over terms, before using them to define the ECTA data structure. In the following, we are interested in equalities between an arbitrary number $n > 0$ of paths rather than just two paths; we refer to such n-ary constraints as *path equivalences classes* (PECs).

Definition 3.1 (Path Equivalence Classes). A *path equivalence class* (PEC) c, is a set of paths. We write a PEC $\{p_1, p_2, \ldots, p_n\}$ as $\{p_1 = p_2 = \cdots = p_n\}$.

Intuitively, the constraint $0.0 = 1.0$ is satisfied on a term t if $t|_{0.0} = t|_{1.0}$; this notion generalizes straightforwardly to non-binary PECs:

[5]The only other solution to this synthesis problem is $h\ y$, which is discovered after backtracking and picking h over g.

Definition 3.2 (Satisfaction of a PEC, Value at a PEC). A path equivalence class $c = \{p_1 = \cdots = p_n\}$ is *satisfied* on a term t if there is some t' such that, $\forall p_i \in c, t|_{p_i} = t'$. We write $t \models c$ if this condition holds, and $t|_c$ to denote this unique t'.

Finally, we discuss sets of PECs, called *path constraint sets* (PCSs):

Definition 3.3 (Path Constraint Sets, Satisfaction, Consistency). A *path constraint set* $C = \{c_1, \ldots, c_m\}$ is a set of disjoint path equivalence classes. A term t satisfies C, written $t \models C$, if $\forall c \in C, t \models c$. If there exists a t such that $t \models C$, then C is *consistent*; otherwise, it is *inconsistent*.

Note that any set of PECs can be *normalized* into a PCS by merging non-disjoint PECs; for example, the set $\{\{0 = 1\}, \{1 = 2\}\}$ can be normalized into $\{\{0 = 1 = 2\}\}$. In the following, we assume that the results of all PCS operations (*e.g.* $C_1 \cup C_2$) are always implicitly normalized.

We are interested in detecting inconsistent PCSs because ECTA operations can use this property to prune empty subautomata. For a single PEC c, consistency is rather straightforward: c is consistent iff it is prefix-free.[6] A non-prefix-free PEC, such as $1.0.0 = 1$, requires a term to be equal to its subterm, which is impossible since terms are finite trees. For a PCS, however, the story is more complicated: in particular, it is not sufficient that each of its member PECs is prefix-free, because two PECs may reference subterms of each other. For example, consider the PCS $C = \{c_1, c_2\} = \{\{0 = 1.0\}, \{0.0 = 1\}\}$. Although c_1 and c_2 are prefix-free, together they imply an inconsistent constraint $1.0.0 = 1$, which can be obtained by substituting 1.0 for 0 in c_2, as justified by c_1.

For more intuition, consider two patterns $f(A, g(A))$ and $f(g(B), B)$; it is easy to see that the terms matching these patterns satisfy the PECs c_1 and c_2, respectively. The conjunction of the two PECs corresponds to the unification of the two patterns, which produces unification constraints $A = g(B)$ and $g(A) = B$, and eventually the contradictory constraint $B = g(g(B))$—which corresponds exactly to the $1 = 1.0.0$ PEC above. In unification parlance, we say that this constraint fails an *occurs check*. Checking consistency of a PCS is the name-free analogue of the occurs check.

Checking Consistency via Congruence Closure. These observations suggest an algorithm for checking PCS consistency: (1) saturate the PCS with all implied equalities (such as $1.0.0 = 1$ above), and (2) check if any of them is non-prefix-free. To formalize the former step, we first declaratively define the *closure* operation on PCSs, and then discuss how to implement it efficiently.

Definition 3.4 (Closure). A PCS C is *closed* if the following holds for any $c_1, c_2 \in C$: for any paths p, p', p'', if $p', p'' \in c_1$ and $p'.p \in c_2$, then $p''.p \in c_2$. In other words, whenever c_2 contains an extension of a path in c_1, it also contains the same extension of *all* paths in c_1. The *closure* of C, denoted $\mathrm{cl}(C)$, is the smallest closed PCS that contains C.

For example, the PCS $C = \{c_1, c_2\} = \{\{0 = 1.0\}, \{0.0 = 1\}\}$ is not closed: if we set $p' = 0, p'' = 1.0$, and $p = 0$, then $0 \in c_1$, $1.0 \in c_1$, and $0.0 \in c_2$, but $1.0.0 \notin c_2$. The closure of this PCS $\mathrm{cl}(C) = \{c_1', c_2'\}$, where c_1' and c_2' are infinite PECs of the form $c_1' = \{0 = 1.0 = 0.0.0 = 1.0.0.0 = \ldots\}$ and $c_2' = \{1 = 0.0 = 1.0.0 = 0.0.0.0 = \ldots\}$.

Theorem 3.5 (Correctness of Closure). *For any term $t \in \mathcal{T}(\Sigma)$, $t \models C \Leftrightarrow t \models \mathrm{cl}(C)$.*

Theorem 3.6 (Consistency of a Closed PCS). *Let C be a closed PCS. Then C is inconsistent iff one of the $c_i \in C$ is not prefix-free.*

Together Theorem 3.5 and Theorem 3.6 ensure the correctness of our consistency checking procedure; what is left is to implement the closure computation efficiently. It turns out this can be done using the well-known congruence closure algorithm for the first-order theory of equality and uninterpreted functions [Nelson and Oppen 1980]. This algorithm finitely

[6] Technically, we must also ensure that $\forall i \in c.i < \max_{s \in \Sigma} \mathrm{arity}(s)$, but this is trivially maintained by all ECTA operations.

Syntax

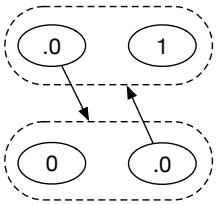
$$
\begin{array}{rll}
c ::= & \{p_1 = \cdots = p_n\} & \text{path equivalence classes} \\
C ::= & \{c_1, \ldots, c_m\} & \text{path constraint sets} \\
n ::= & \mathbf{U}(\overline{e}) & \text{nodes (states)} \\
e ::= & \mathbf{\Pi}(s, \overline{n}, C) \mid \mathbf{\Pi}_\perp & \text{transitions}
\end{array}
$$

Denotation

$$
\begin{array}{rl}
[\![\mathbf{U}(\overline{e})]\!]^N & = \bigcup_i [\![e^i]\!]^E \\
[\![\mathbf{\Pi}(s, \overline{n}, C)]\!]^E & = \left\{ s(\overline{t}) \;\middle|\; t^i \in [\![n^i]\!]^N, s(\overline{t}) \models C \right\}
\end{array}
$$

Fig. 5. Acyclic ECTAs: syntax and semantics. Here $s \in \Sigma$ and p is a path.

represents a possibly infinite set of equalities using an e-graph. Hence, to check consistency of a PCS C, we can simply (1) add each path of C into an e-graph, interpreting path prefixes as subterms (*i.e.* 1.0 is .0 applied to 1); (2) merge all paths from the same PEC into one e-class and run congruence closure on the e-graph; (3) check if the resulting e-graph has cycles; if so, then C is inconsistent. The figure on the right shows the (cyclic) e-graph obtained by running this algorithm on our example $\{\{0 = 1.0\}, \{0.0 = 1\}\}$.



3.3 Acyclic ECTAs: Core Definition

Like string automata, tree automata are usually formalized as graphs, defined by a set of states and a transition function. For our purposes, it is more convenient to formalize ECTAs using a recursive grammar, in the same style VSAs are typically presented [Polozov and Gulwani 2015].

Syntax. Fig. 5 (left) shows the grammar for acyclic ECTAs, consisting of mutually recursive definitions for nodes (states) $n \in N$ and transitions $e \in E$; an ECTA then is identified with its root node, which represents the final state.[7] In a transition $\mathbf{\Pi}(s, \overline{n}, C)$,[8] the number of child nodes $|\overline{n}|$ must equal arity(s); both $\overline{n}$ and C can be omitted when empty. As is common for VSAs, we assume implicit sharing of sub-trees: that is, an acyclic ECTA is a DAG with no duplicate sub-graphs.

The special symbol $\mathbf{\Pi}_\perp$ denotes an "empty transition", which is used in intermediate results of ECTA operations. For symmetry, we also abbreviate the empty node, $\mathbf{U}()$, as $\mathbf{U}_\perp$. A *normalized* ECTA contains no occurrences of $\mathbf{\Pi}_\perp$ or $\mathbf{U}_\perp$, unless the root is itself $\mathbf{U}_\perp$. Any ECTA can be normalized by iteratively replacing any transition containing a $\mathbf{U}_\perp$ child with $\mathbf{\Pi}_\perp$, and removing all instances of $\mathbf{\Pi}_\perp$ from the children of each node. For instance, $\mathbf{U}(\mathbf{\Pi}(\mathrm{a}), \mathbf{\Pi}(+, [\mathbf{U}(\mathbf{\Pi}(\mathrm{b})), \mathbf{U}_\perp]))$ normalizes to $\mathbf{U}(\mathbf{\Pi}(\mathrm{a})))$. We assume henceforth that all ECTAs are implicitly normalized after every operation.

Semantics and Spurious Runs. The *denotation* of an acyclic ECTA, *i.e.* the set of terms it accepts, is defined in Fig. 5 (right) as a pair of mutually-recursive functions: $[\![\cdot]\!]^N : N \to \mathbb{P}(\mathcal{T}(\Sigma))$ and $[\![\cdot]\!]^E : E \to \mathbb{P}(\mathcal{T}(\Sigma))$. We define a partial order $\prec$ on ECTAs as the subset order on their denotations: $n_1 \prec n_2$ iff $[\![n_1]\!]^N \subseteq [\![n_2]\!]^N$. The *skeleton* of an ECTA, sk(n), is obtained by recursively removing all path constraints from its transitions. A *spurious run* of n is a term t, that is rejected by n but accepted by its skeleton: $t \notin [\![n]\!]^N \wedge t \in [\![\mathrm{sk}(n)]\!]^N$.

3.4 Basic Operation: Union and Intersection

We now present algorithms for two basic operations on ECTAs, union and intersection. They serve as building blocks for our two core contributions: static and dynamic reduction.

Union. The union of two ECTAs, $n_1 \sqcup n_2$, simply merges the transition of their root nodes:

Definition 3.7 (Union). Let $n_1 = \mathbf{U}(\overline{e_1}), n_2 = \mathbf{U}(\overline{e_2})$ be two nodes. Then $n_1 \sqcup n_2 = \mathbf{U}(\overline{e_1} \cup \overline{e_2})$.

[7] Although this representation is restricted to ECTAs with a single final state (the root node), this is not an important restriction: any acyclic tree automaton is equivalent to the same automaton with all its final states merged into one.
[8] Hereafter we write $\overline{x}$ to denote a sequence of xs, with x^i referring to the i-th element of that sequence.

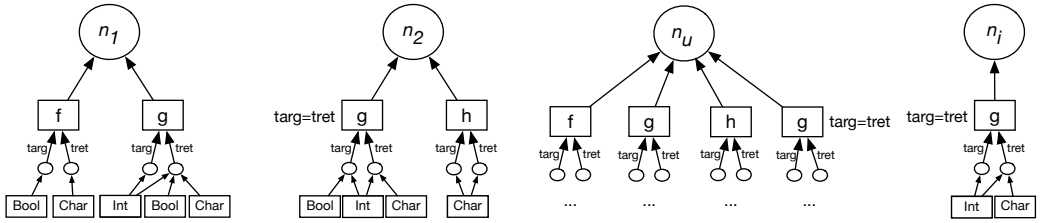


Fig. 6. Two ECTAs n_1 and n_2, their union n_u and intersection n_i.

Fig. 6 gives an example of ECTA union $n_u = n_1 \sqcup n_2$.

Theorem 3.8 (Correctness of ECTA Union). $[\![n_1 \sqcup n_2]\!]^N = [\![n_1]\!]^N \cup [\![n_2]\!]^N$.

Intersection. The intersection of two ECTAs is more involved. Intersecting two nodes, $n_1 \sqcap n_2$, involves intersecting all pairs of their transitions; intersecting two transitions, $e_1 \sqcap e_2$, in turn, involves intersecting their child nodes point-wise, and is only well-defined if the symbols and PCSs of e_1 and e_2 are compatible:

Definition 3.9 (Intersection). Let $n_1 = \mathbf{U}(\overline{e_1}), n_2 = \mathbf{U}(\overline{e_2})$ be two nodes, then:

$$n_1 \sqcap n_2 = \mathbf{U}\left(\left\{ e_1^i \sqcap e_2^j \;\middle|\; e_1^i \in \overline{e_1}, e_2^j \in \overline{e_2} \right\}\right)$$

Let $e_1 = \mathbf{\Pi}(s_1, [n_1^0 \ldots n_1^{k-1}], C_1)$, $e_2 = \mathbf{\Pi}(s_2, [n_2^0 \ldots n_2^{l-1}], C_2)$ be two transitions, then:

$$e_1 \sqcap e_2 = \begin{cases} \mathbf{\Pi}(s_1, [n_1^0 \sqcap n_2^0, \ldots, n_1^{k-1} \sqcap n_2^{k-1}], C_1 \cup C_2) & \text{if } s_1 = s_2 \text{ and } C_1 \cup C_2 \text{ is consistent} \\ \mathbf{\Pi}_\perp & \text{otherwise} \end{cases}$$

Consider the example of ECTA intersection $n_i = n_1 \sqcap n_2$ in Fig. 6. To compute the intersection at the top level, we intersect all pairs of transitions—(f, g), (f, h), (g, g), and (g, h)—but the three pairs with incompatible function symbols simpy yield $\mathbf{\Pi}_\perp$ and are discarded. To intersect the two g-transitions, we recursively intersect their `targ` and `tret` nodes; the resulting g-transition also inherits its constraint from n_2.

Theorem 3.10 (Correctness of ECTA Intersection). $[\![n_1 \sqcap n_2]\!]^N = [\![n_1]\!]^N \cap [\![n_2]\!]^N$.

Proposition 3.11. $\mathbf{U}_\perp \sqcap n = n \sqcap \mathbf{U}_\perp = \mathbf{U}_\perp$

Corollary 3.12. *Define $n_1 \cong n_2$ if $[\![n_1]\!]^N = [\![n_2]\!]^N$. Then, with respect to $(\cong)$, the $(\sqcap)$ and $(\sqcup)$ operations form a distributive lattice, with $\mathbf{U}_\perp$ as the bottom element, and $(\prec)$ as the order.*

3.5 Static Reduction

We are now ready to present static reduction, the first of the two core algorithms that enable efficient extraction of terms satisfying ECTA constraints. Consider the example in Fig. 2. Intuitively, the constraint `fun.targ = arg.type` has been *reduced* in Fig. 2b, because with f elimitated, *every* possible formal parameter type at path `fun.targ` matches *some* actual parameter type at path `arg.type`. More generally, a binary constraint $p_1 = p_2$ is reduced if everything at path p_1 matches something at path p_2; this definition extends naturally to non-binary constraints. We now define the machinery to state this formally, and then provide a simple algorithm for reducing a constraint, which builds upon ECTA intersection.

Subautomaton at a Path. First, we generalize the definition of a subterm at a path, $t|_p$, to ECTAs:

Definition 3.13 (Nodes at path, Subautomaton at a path). The set $\mathsf{nodes}(n, p)$ of nodes reachable from $n = \mathbf{U}(\overline{e})$ via path p is defined as:

$$\mathsf{nodes}(n, \epsilon) = \{n\} \qquad \mathsf{nodes}(n, j.p) = \bigsqcup_i \mathsf{nodes}(e^i, j.p)$$

The set $\mathsf{nodes}(e, p)$ of nodes reachable from a transition $e = \mathbf{\Pi}(s, \overline{n}, C)$ is defined as:

$$\mathsf{nodes}(e, j.p) = \begin{cases} \mathsf{nodes}(n^j, p) & j < \mathsf{arity}(s) \\ \emptyset & \text{otherwise} \end{cases}$$

Finally, the *subautomaton* of n at path p is defined as $n|_p = \bigsqcup \mathsf{nodes}(n, p)$; similarly, the subautomaton of e is defined as $e|_p = \bigsqcup \mathsf{nodes}(e, p)$.

In Fig. 2a, if n is the root node, then $\mathsf{nodes}(n, \texttt{arg.type}) = \{\mathbf{U}(\mathbf{\Pi}(\texttt{Int})), \mathbf{U}(\mathbf{\Pi}(\texttt{Char}))\}$ and $n|_{\texttt{arg.type}} = \mathbf{U}([\mathbf{\Pi}(\texttt{Int}), \mathbf{\Pi}(\texttt{Char})])$. The reader might be wondering why define $\mathsf{nodes}(e, j.p) = \emptyset$ for an out-of-bounds index j instead of restricting these and following definitions to "well-formed" paths. The rationale is to enable ECTAs to have "cousin" transitions with different arities, and be able to navigate to nodes and subautomata at higher arities, by simply discarding branches with lower arities; this flexibility is required, for instance, in our full encoding of type-driven synthesis in §7.2.

Without equality constraints, the denotation of $n|_p$ would simply be the set of subterms $t|_p$ of all terms t represented by n. With equality constraints, $n|_p$ is an overapproximation of that set, since the equality constraints on the topmost layers get ignored.

Lemma 3.14 (Correctness of subautomaton at a path). *For any n, p, $[\![n|_p]\!]^N \supseteq \left\{ t|_p \;\middle|\; t \in [\![n]\!]^N \right\}$. Similarly, for any e, $[\![e|_p]\!]^E \supseteq \left\{ t|_p \;\middle|\; t \in [\![e]\!]^E \right\}$.*

This lemma can be used to prove that a term is present in $[\![n|_p]\!]^N$ but not that it is absent. Fortunately, we only need to show presence when proving soundness of static reduction (Theorem 3.20).

Reduction Criterion. We can now formally state what it means for a constraint to be *reduced*:

Definition 3.15 (Reduction Criterion). Let $e = \mathbf{\Pi}(s, \overline{n}, C)$ be a transition and let $c = \{p_1 = \cdots = p_k\} \in C$. We say that e satisfies the *reduction criterion* for c (alternatively, c is *reduced* at e) if, for each $p_i, p_j \in c$ and each $n \in \mathsf{nodes}(e, p_i)$, $n \sqcap e|_{p_j} \neq \mathbf{U}_\perp$.

The reduction criterion suggests an algorithm for reducing a path constraint: given a constraint $p_1 = p_2$ on transition e, replace every node n reachable via p_1 with $n \sqcap e|_{p_2}$. As a result, every node in $\mathsf{nodes}(e, p_1)$ will match some node in $\mathsf{nodes}(e, p_2)$. For example, to reduce the constraint `fun.targ = arg.type` at the transition `app` in Fig. 2, the algorithm first computes $\texttt{app}|_{\texttt{arg.type}}$, the automaton representing all possible actual parameter types; the result is $n_a = \mathbf{U}(\mathbf{\Pi}(\texttt{Int}), \mathbf{\Pi}(\texttt{Char}))$. Next, it intersects n_a it with each of the three nodes reachable via `fun.targ`, that is, `Int`, `Char`, and `Bool`. This has no effect on the `targ` children of g and h, but the `targ` child of f becomes $\mathbf{U}_\perp$, leading to the removal of the f transition upon normalizatoin and resulting in Fig. 2b.

Intersection at a Path. In order to formalize the reduction algorithm outlined above, we introduce the notion of *intersection at a path*.

Definition 3.16 (Intersection at a Path). Intersecting node n with node n' at path p, denoted $n|_p^{\sqcap n'}$, replaces all nodes reachable from n via p with their intersection with n'. More formally, if $n = \mathbf{U}(\overline{e})$:

$$n|_\epsilon^{\sqcap n'} = n \sqcap n' \qquad n|_{j.p}^{\sqcap n'} = \mathbf{U}(e^i|_{j.p}^{\sqcap n'})$$

where intersecting a transition $e = \mathbf{\Pi}(s, [n^0, \ldots, n^{k-1}], C)$ at a non-empty path p is defined as:

$$e|^{\sqcap n'}_{j.p} = \begin{cases} \mathbf{\Pi}(s, [n^0, \ldots, n^j|^{\sqcap n'}_p, \ldots, n^{k-1}], C) & j < \text{arity}(s) \\ \mathbf{\Pi}_\bot & \text{otherwise} \end{cases}$$

For example, in Fig. 2a, intersecting the root node n at path `fun.targ` with the node n_a from our previous example ($n_a = \mathbf{U}(\mathbf{\Pi}(\texttt{Int}), \mathbf{\Pi}(\texttt{Char}))$) yields the ECTA in Fig. 2b.

Lemma 3.17. *$t \in [\![\left(n|^{\sqcap n'}_p\right)|_p]\!]^N$ if and only if $t \in [\![n|_p]\!]^N$ and $t \in [\![n']\!]^N$. Similarly, $t \in [\![\left(e|^{\sqcap n'}_p\right)|_p]\!]^E$ if and only if $t \in [\![e|_p]\!]^N$ and $t \in [\![n']\!]^N$.*

Reduction Algorithm. With this new terminology, we can recast our previous explanation of how the constraint `fun.targ = arg.type` in Fig. 2 gets reduced: once we have obtained the "actual parameter automaton" $n_a = \texttt{app}|_{\texttt{arg.type}}$, we can simply return $n|^{\sqcap n_a}_{\texttt{fun.targ}}$ (where n is the root node). This explanation needs one final tweak: in this example, the information only propagates in one direction—from `arg.type` to `fun.targ`—because the types of the actuals happen to be a subset of the types of the formals; in general, though, reduction needs to propagate information both ways. Hence a more accurate recipe for how to perform the reduction in Fig. 2 is: (1) compute the automaton $n^* = (n|_{\texttt{fun.targ}}) \sqcap (n|_{\texttt{arg.type}})$, capturing all *shared* formal and actual parameter types; (2) intersects the root with n^* at *both* paths involved in the constraint: $\left(n|^{\sqcap n^*}_{\texttt{fun.targ}}\right)|^{\sqcap n^*}_{\texttt{arg.type}}$. We extrapolate this description into a general algorithm for static reduction:

Definition 3.18 (Static Reduction). Let $c = \{p_1 = \cdots = p_k\}$ be a prefix-free PEC; then

$$\text{reduce}(e, c) = e|^{\sqcap n^*}_{p_1} \ldots |^{\sqcap n^*}_{p_k} \qquad \text{where} \quad n^* = \bigsqcap_{p_i \in c} e|_{p_i}$$

Theorem 3.19 (Completeness of Reduction). *$\text{reduce}(e, c)$ satisfies the reduction criterion for c.*

Theorem 3.20 (Soundness of Reduction). *Let $e = \mathbf{\Pi}(s, \overline{n}, C)$ be a transition and $c \in C$; then:*

$$[\![\text{reduce}(e, c)]\!]^E = [\![e]\!]^E$$

4 FAST ENUMERATION WITH DYNAMIC REDUCTION

We now turn to our second core contribution: the algorithm for efficiently extracting (or enumerating) terms represented by an ECTA. As we have outlined in §2.4, the main idea behind the algorithm is to avoid eager enumeration of constrained nodes, instead replacing them with "unification" variables—the mechanism we dub *dynamic reduction*.

Inspired by presentations of DPLL(T) and Knuth-Bendix completion [Bachmair and Dershowitz 1994; Nieuwenhuis et al. 2006], we formalize the enumeration algorithm as a non-deterministic transition system. Configurations of this system are called *enumeration states* and steps are governed by two rules, Choose and Suspend. Intuitively, Choose handles unconstrained ECTA nodes, making a non-deterministic choice between their incoming transitions; Suspend handles constrained nodes, suspending them into variables. Fig. 7, which serves as the running example for this section, shows an example sequence of Choose and Suspend steps applied to a simplified version of the ECTA from Fig. 4 (the simplified ECTA encodes all well-typed size-two terms in the environment $\Gamma = \{x\colon \texttt{Int}, y\colon \texttt{Char}, g\colon \alpha \to \alpha, h\colon \texttt{Char} \to \texttt{Bool}\}$).

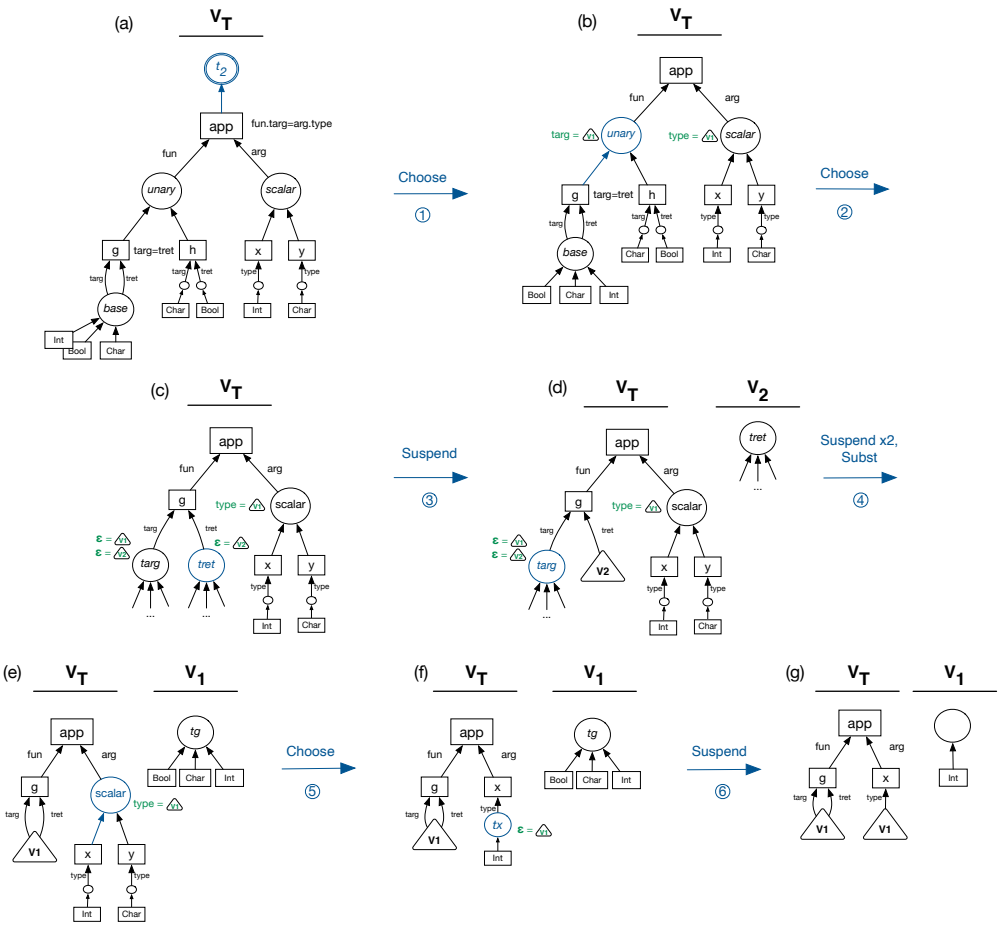


Fig. 7. An example sequence of steps through enumeration states. The focus node of each step is highlighted in blue and constraint fragments are highlighted in green. The final state is fully enumerated.

4.1 Enumeration State

The syntax of enumeration states is shown in Fig. 8 (left). Var is a countably infinite set of variables, with a dedicated "root" variable $v_\top \in \text{Var}$. An *enumeration state* σ is a mapping from variables to *partially-enumerated terms* (or *p-terms* for short). With the exception of $v_\top$, which stores the top-level enumeration result, each variable captures a set of ECTA nodes that are constrained to be equal. For example, in Fig. 7 (g), the variable v_1 captures the nodes `g.targ` and `x.type`, which are equated by the constraint on `app`, and also `g.tret`, equated to the former by the constraint on `g`.

A p-term τ is a term that might contain variables and *unenumerated nodes* (*u-nodes* for short). A u-node $\square(n, \Phi)$ is an ECTA node n annotated with zero or more *constraint fragments* ϕ, each consisting of a PEC and a variable. Intuitively, a constraint fragment is a constraint propagated downward from an ECTA transition. For example, in Fig. 7 (b), when the original constraint `fun.targ = arg.type` on `app` is propagated down to *unary* and *scalar*, it is split into two fragments: $\langle \texttt{targ} = v_1 \rangle$ and $\langle \texttt{type} = v_1 \rangle$. The splitting is necessary because for each of the child u-nodes one of the sides of this constraint is "out of scope"; hence a fresh variable v_1 is introduced to refer to

Enumeration States

$s \in \Sigma, v \in \mathsf{Var}, c \in \mathsf{PEC}$

$\phi ::=$	$\langle c = v \rangle$	*Constraint fragments*
$\Phi ::=$	$\overline{\phi}$	*Cons. fragment sets*
$\tau ::=$		*P-terms*
	$\mid v \mid s(\overline{\tau})$	variable, application
	$\mid \square(n, \Phi)$	unenumerated node
$\sigma ::=$	$[\overline{v \mapsto \tau}]$	*Enumeration states*
$C[\cdot] ::=$		*Contexts*
	$\mid \cdot$	
	$\mid s(\overline{\tau}, C[\cdot], \overline{\tau})$	

Enumeration step $\boxed{\tau \longrightarrow \tau', \sigma \longrightarrow \sigma'}$

$$\textsc{Choose-}\square\ \frac{\langle \epsilon = \_ \rangle \notin \Phi \quad \mathbf{\Pi}(s, [n^0, \ldots, n^{k-1}], C) \in \overline{e} \quad C' = \{\langle c^j = v^j \rangle \mid c^j \in C, v^j \text{ is fresh}\} \quad \tau^i = \square(n^i, \mathsf{project}(C' \cup \Phi, i))}{\square(\mathbf{U}(\overline{e}), \Phi) \longrightarrow s(\tau^0, \ldots, \tau^{k-1})}$$

$$\textsc{Choose}\ \frac{\sigma[v] = C[\tau] \quad \tau \longrightarrow \tau' \quad v \text{ is solved}}{\sigma \longrightarrow \sigma[v \mapsto C[\tau']]}$$

$$\textsc{Suspend-1}\ \frac{\sigma[v] = C[\square(n, \langle \epsilon = v' \rangle \cup \Phi)] \quad v' \notin \mathrm{dom}(\sigma)}{\sigma \longrightarrow \sigma[v \mapsto C[v'], v' \mapsto \square(n, \Phi)]}$$

$$\textsc{Suspend-2}\ \frac{\sigma[v] = C[\square(n, \langle \epsilon = v' \rangle \cup \Phi)] \quad \sigma[v'] = \square(n', \Phi')}{\sigma \longrightarrow \sigma[v \mapsto C[v'], v' \mapsto \square(n \sqcap n', \Phi \cup \Phi')]}$$

Fig. 8. Enumeration states and rules.

the common value at both sides. A variable v is *solved* in σ iff it is not mentioned in any of the constraint fragments; for example, v_1 is unsolved in Fig. 7 (b)–(f) and solved in Fig. 7 (g).

A u-node is *restricted* iff its Φ is non-empty; an unrestricted u-node is written $\square(n)$. An enumeration state σ is called *fully enumerated* if there are no restricted u-nodes anywhere inside σ. The reader might be surprised that a fully enumerated state is allowed to have u-nodes at all; as we explain in §4.4, this enables compact representation of enumeration results with "trivial differences."

Denotation. The denotation of an enumeration state $[\![\sigma]\!]^S$ is a set of substitutions $\rho\colon \mathsf{Var} \rightharpoonup \mathcal{T}(\Sigma)$, which is compatible with the constraint fragments and subterm relations imposed by variables inside p-terms. Because of the circular dependencies between a p-term and its enclosing σ, the formal definition is somewhat technical and therefore relegated to the extended version.

4.2 Enumeration Rules

Fig. 8 (right) formalizes the above-mentioned Choose and Suspend rules as a step relation $\sigma \longrightarrow \sigma$ over enumeration states and an auxiliary step relation $\tau \longrightarrow \tau'$ over p-terms.

Choose. We first formalize the auxiliary rule Choose-□ for p-terms. This rule takes a u-node, non-deterministically selects one of its transitions e, and steps to a p-term that has e's function symbol at the root and new u-nodes as children. Step ⑤ in Fig. 7 is an example application of this rule: here the original u-node *scalar* turns into one of its two incoming transitions, `x`; step ① is also an instance of this rule, albeit with no alternatives.

The tricky aspect of Choose-□ is propagating constraints—either C from the transition e or Φ from the original u-node—to the newly minted u-nodes. The former scenario is illustrated in step ①: here the PEC $c = \{\texttt{fun.targ} = \texttt{arg.type}\}$ on the `app` transition is split into two fragments, $\langle \texttt{targ} = v_1 \rangle$ and $\langle \texttt{type} = v_1 \rangle$, attached to the new u-nodes *unary* and *scalar*, respectively. To this end, Choose-□ first creates a fresh variable v_1 and forms a constraint fragment $\langle c = v_1 \rangle$ using the original PEC c; next it *projects* this fragment down to each i-th child, retaining only those paths of c that start with i and chopping off their heads. The project function is defined formally in Fig. 9. Note how the two new fragments together completely capture the original constraint.

The latter scenario—propagating existing constraint fragments—is illustrated in step ⑤. Here the u-node *scalar* is restricted by the fragment $\langle \texttt{type} = v_1 \rangle$; in this case, there is no need to create new variables: the existing fragment is simply projected down to the child *tx* and becomes $\langle \epsilon = v_1 \rangle$. In the general case, both new and existing constraint fragments should be combined; this is the

case in step ②, where the u-node *targ* inherits the fragment $\langle \epsilon = v_1 \rangle$ from *unary*, and also acquires a new fragment $\langle \epsilon = v_2 \rangle$ by splitting the constraint on `g`.

Finally, consider the rule Choose, which lifts Choose-□ to whole enumeration states. This rule allows making a step inside any component of σ, as long as its variable is solved. For example, in Fig. 7 (e) we are not allowed to make a step inside v_1 (say, choosing `Bool` among the three types), because v_1 still appears in the constraint fragment $\langle \texttt{type} = v_1 \rangle$ inside $v_\top$. The rationale for this restriction is to avoid making premature choices for constrained nodes: in our example, picking `Bool` would be a mistake, which is entirely avoidable by simply waiting until all constraints are resolved (such as the state in Fig. 7 (g)).

Suspend. The Suspend rules handle u-nodes with *ϵ-fragments*, *i.e.* constraint fragments of the form $\langle \epsilon = v \rangle$.[9] Intuitively, an ϵ-fragment indicates that this node is the target of a constraint captured by v. In response, the Suspend rules simply "move" the target u-node to the v-component of the state, replacing it with v in the original p-term.

The two Suspend rules differ in whether the current state σ already has a mapping for v: if it does not, Suspend-1 initializes this mapping with its target u-node $\square(n, \Phi)$; if it does, Suspend-2 updates this mapping, combining the old u-node $\square(n', \Phi')$ and the new one $\square(n, \Phi)$ by intersecting their ECTAs and merging their constraint fragments. Note that the old value of v must necessarily be a u-node, because Choose is not allowed to operate under unsolved variables.

An example application of Suspend-1 is step ③ of Fig. 7. The target node *tret* has an ϵ-fragment $\langle \epsilon = v_2 \rangle$; since v_2 is uninitialized, Suspend-1 creates a new mapping $[v_2 \mapsto \mathit{tret}]$. Step ⑥, on the other hand, is an example of Suspend-2: the target node *tx* is restricted by $\langle \epsilon = v_1 \rangle$; since v_1 already maps to *tg*, Suspend-2 updates it with $\mathit{tx} \sqcap \mathit{tg}$. As a result of this intersection, v_1 now contains only those types (in this case, the sole type `Int`) that make the term represented by $v_\top$ well-typed.

Eliminating Redundant Variables. Finally, let us demystify the transformation ④ in Fig. 7, which consists of three atomic steps. The first step suspends *targ*, which has *not one but two* ϵ-fragments—$\langle \epsilon = v_1 \rangle$ and $\langle \epsilon = v_2 \rangle$—either of which can be targeted by a Suspend. Suppose that the second one is chosen (both choices lead to equivalent results, up to variable renaming). Since v_2 is already initialized, Suspend-2 fires, merging *tret* and *targ* into a single u-node *tg'* under v_2; importantly, *tg'* inherits the other constraint fragment from *targ*, namely $\langle \epsilon = v_1 \rangle$. Because of that, Suspend-1 can now fire on *tg'*, creating the state $[v_\top \mapsto \ldots, v_2 \mapsto v_1, v_1 \mapsto \mathit{tg}]$, where *tg* is *tg'* stripped of its constraint fragment. This new state is a bit awkward, since it contains a "redundant" variable v_2, which simply stores another variable, v_1. To get rid of such redundant variables, we introduce an auxiliary rule Subst, which simply replaces all occurrences of v_2 with v_1 and removes the unused mapping from σ (see Fig. 9). After applying Subst, we arrive at the state in Fig. 7 (e).

4.3 Enumeration Algorithm

We are now ready to describe the top-level algorithm Enumerate. The algorithm takes as input an ECTA n and produces a stream of fully-enumerated states. To this end, it first creates an initial state $\sigma_0 = [v_\top \mapsto \square(n)]$, and then enumerates derivations of $\sigma_0 \longrightarrow^* \sigma_\bullet$ where $\sigma_\bullet$ is fully enumerated and $\longrightarrow^*$ is the reflexive-transitive closure of $\longrightarrow$. In each step, the algorithm has the freedom to select (i) which u-node to target, and (ii) in the case of Choose, which transition to choose. The enumeration rules are designed in such a way that the former selection constitutes "don't care non-determinism" (*i.e.* any target node can be selected without loss of completeness); this is in contrast to the latter selection, which constitutes "don't know non-determinism" and must be backtracked. At the same time, different schedules of rule applications might lead to significantly

[9] Choose-□ does not apply to these nodes thanks to its first premise. Note also that because all PECs in the original ECTA are prefix-free and this property is maintained by project, any fragment that contains ϵ, must *only* contain ϵ.

Projecting constraint fragments

$$\text{project}(\Phi, i) = \bigcup_{\phi \in \Phi} \text{project}(\phi, i)$$

$$\text{project}(\langle c = v \rangle, i) = \begin{cases} \{\langle c' = v \rangle\} & \text{if } c' \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases}$$

$$\text{where } c' = \bigcup_{p \in c} \text{project}(p, i)$$

$$\text{project}(p, i) = \begin{cases} \bot & \text{if } p = \epsilon \\ \{p'\} & \text{if } p = i.p' \\ \emptyset & \text{otherwise} \end{cases}$$

Enumeration step (cont.) $\boxed{\sigma \longrightarrow \sigma'}$

$$\textsc{Subst}\ \frac{\sigma[v_2] = v_1}{\sigma \longrightarrow [v_1/v_2]\,(\sigma \setminus [v_2 \mapsto v_1])}$$

Fig. 9. Auxiliary definitions

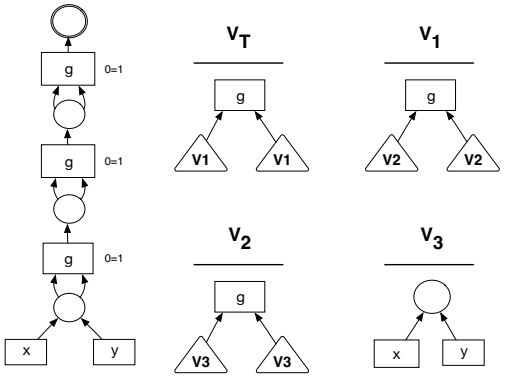


Fig. 10. (a) An ECTA representing all perfect trees of depth three, whose leaves are either all x or all y (b) The ECTA fully enumerated, in logarithmic space

different performance. The ECTA library provides a default schedule—depth-first, left to right—but enables the user to specify a domain-specific schedule in order to optimize performance.

Theorem 4.1 (Termination of Enumeration). *There is no infinite sequence* $\sigma_0 \longrightarrow \sigma_1 \longrightarrow \ldots$.

Theorem 4.2 (Correctness of Enumeration). *Let* n *be an ECTA,* σ_0 *be the initial enumeration state, and consider all finite sequences* $\sigma_0 \longrightarrow^* \sigma_\bullet$*, such that* $\sigma_\bullet$ *is fully enumerated; then:*

$$[\![n]\!]^N = \left\{ \rho(v_\top) \;\middle|\; \sigma_0 \longrightarrow^* \sigma_\bullet, \rho \in [\![\sigma_\bullet]\!]^S \right\}$$

4.4 Compact Fully Enumerated States

We now return to the design decision to allow (unrestricted) u-nodes in fully enumerated states. Our running example in Fig. 7 does not motivate this decision very well: the fully enumerated state in Fig. 7 (g) encodes a single term anyway, so it seems only natural to let Choose loose on the last remaining u-node. For other ECTAs, however, a single fully-enumerated state might represent exponentially many[10] terms, or the terms might be exponentially larger, or both. For an example, consider Fig. 10a. This ECTA represents the set of all perfect binary trees of depth three, whose leaves are either all x or all y. A moment's thought reveals that this set contains two trees, each of size 15. Instead of returning these two large trees explicitly, the fully-enumerated state $\sigma_\bullet$ in Fig. 10b represents them as a *hierarchy of unconstrained tree automata*, from which the concrete trees may be trivially generated. It is straightforward to see that the sizes of the two perfect trees grow exponentially with their depth, while the size of $\sigma_\bullet$ grows only linearly.

The main benefit of this design, however, is that, depending on the problem domain, some nodes *need not be enumerated at all*, as long as we know their denotation is non-empty. For example, to determine whether a propositional formula is satisfiable (§7.1), it is often enough to provide a *partial satisfying assignment*, because the values of the unassigned variables are irrelevant; such a partial assignment can be represented by a $\sigma_\bullet$, where irrelevant variables are left unenumerated. Similarly, in type-driven synthesis, the polymorphic type of a component need not always be fully instantiated, as long as we know that a compatible instantiation exists. In fact, as we explain in §7.2,

[10] Or, with the cyclic ECTAs of §5, infinitely many.

Syntax

$$
\begin{array}{rll}
n ::= & \mathbf{U}(\overline{e}) \mid \mu x.\mathbf{U}(\overline{e}) \mid x & \text{nodes (states)} \\
e ::= & \mathbf{\Pi}(s, \overline{n}, C) \mid \mathbf{\Pi}_{\bot} & \text{transitions}
\end{array}
$$

Unfolding Recursive Nodes

$$
\begin{array}{rl}
\text{unfold}(\mu x.n) & = [\mu x.n/x]\, n \\
\text{unfold}(\mathbf{U}(\overline{e})) & = \mathbf{U}(\overline{e})
\end{array}
$$

Denotation

$$
\begin{array}{rl}
[\![\mathbf{U}(\overline{e})]\!]^N & = \bigcup_i [\![e^i]\!]^E \\
[\![\mu x.n]\!]^N & = [\![\text{unfold}(\mu x.n)]\!]^N \\
[\![\mathbf{\Pi}(s, \overline{n}, C)]\!]^E & = \left\{ s(\overline{t}) \;\middle|\; t^i \in [\![n^i]\!]^N, s(\overline{t}) \models C \right\}
\end{array}
$$

Fig. 11. Cyclic ECTAs: syntax and semantics. Here $s \in \Sigma$, C is a PCS, and x is a bound variable.

cyclic ECTAs can encode infinitely many possible polymorphic instantiations, and enumerating them all would be simply impossible.

5 CYCLIC ECTA

We now present the formalism for fully general ECTAs, which may contain cycles. With cycles, an ECTA node can now represent an infinite space of terms, such as an arbitrary term in some context-free language, including (as in §7.2) the language of arbitrary Haskell types. While this requires an extension to the syntax of ECTAs to allow recursion, shockingly, none of the algorithms require substantial modification.

5.1 Cyclic ECTAs: Core Definition

We extend acyclic ECTAs to cyclic by adding "recursive nodes" $\mu x.\mathbf{U}(\overline{e})$. Within this node, x is a variable bound to $\mathbf{U}(\overline{e})$. In diagrams, we depict any use of x as a back-edge to $\mathbf{U}(\overline{e})$ and keep the μ binding itself implicit. Semantically, x can be replaced with a copy of the node it is bound to, so that an ECTA n is equivalent to $[\mathbf{U}(\overline{e})/x]n$—or rather, to $[\mu x.\mathbf{U}(\overline{e})/x]n$, since $\mathbf{U}(\overline{e})$ contains further uses of x. Fig. 12a shows an cyclic example ECTA, with a recursive node *Nat* representing arbitrary natural numbers defined by the grammar $\texttt{Nat} ::= S(\texttt{Nat}) \mid Z$. Fig. 11 gives the syntax and semantics of cyclic ECTAs. The recursive definition of $[\![n]\!]^N$ should be interpreted with least-fixed-point semantics (as it may unfold arbitrarily many times). Note that this grammar excludes nodes like $\mu x.x$ or $\mu x.\mu y.x$, which would be meaningless. We again assume implicit sharing of sub-trees.[11]

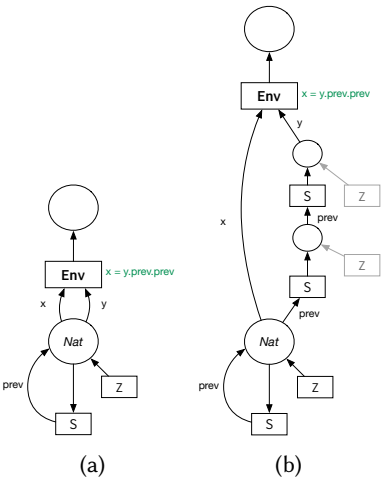


Fig. 12. (a) ECTA representing an environment with two arbitrary natural numbers x and y, where $y = x + 2$. The Nat node is represented $\mu x.\mathbf{U}(\mathbf{\Pi}(S, x), \mathbf{\Pi}(Z))$. (b) The ECTA unfolded into lasso form. The grayed-out transitions will be removed by static reduction.

Cyclic ECTAs become unwieldy when constraints are allowed inside cycles. As a recursive node $\mu x.n$ is repeatedly unfolded and its constraints duplicated, it can yield an arbitrarily large constraint system whose smallest solution may be arbitrarily large. In fact, in this general case, ECTA emptiness is undecidable:

Theorem 5.1 (Undecidability of Emptiness). *Determining whether $[\![n]\!]^N = \emptyset$ is undecidable.*

Proof. By reduction from the Post Correspondence Problem (see extended version). □

[11] However, this pseudo-tree representation precludes sharing of some nodes which would be shared in a true graph representation.

Nodes at Path

$$\mathsf{nodes}(\mu x.n, p) = \mathsf{nodes}(\mathsf{unfold}(\mu x.n), p)$$

Intersection at a Path

$$(\mu x.n)|^{\sqcap p}_{n'} = \mathsf{unfold}(\mu x.n)|^{\sqcap p}_{n'}$$

Enumeration step $\boxed{\tau \longrightarrow \tau'}$

$$\textsc{Choose-}\mu\ \frac{\Phi \neq \emptyset}{\square(\mu x.n, \Phi) \longrightarrow \square(\mathsf{unfold}(\mu x.n), \Phi)}$$

Fig. 13. Extensions to prior algorithms to account for recursive nodes

This motivates a restriction barring constraints on cycles, which guarantees that the constraint system remains finite and efficient algorithms remain possible. More formally:

Definition 5.2 (Finitely-constrained ECTA). An ECTA n is *finitely-constrained* if, for all recursive nodes $\mu x.m$ reachable from n, $m = \mathsf{sk}(m)$.

We assume henceforth that all ECTAs are finitely-constrained. What makes such ECTAs tractable is that, after sufficient unfolding, they enter what we call *lasso form*:

Definition 5.3 (Lasso form). An ECTA n is in *lasso form* if it contains no constrained recursive nodes (*i.e.*, no path constraint references a recursive node).

An ECTA in lasso form is split into a top portion, which contains constraints but no cycles, and a bottom portion, which contains cycles but no constraints. The top portion permits only finitely many choices, while the bottom portion can be enumerated and intersected as in classic tree automata theory. While they may perform intersection on entire subautomata, neither static nor dynamic reduction directly inspect nodes beneath the deepest constraint. Hence, with an updated definition of intersection, our definitions for both static and dynamic reduction work unmodified on ECTAs in lasso form. An example ECTA in lasso form is in Fig. 12b.

5.2 Algorithms for Cyclic ECTAs

Intersection. One formulation of intersection for classic string automata is a depth-first search that begins from a pair of initial or final states and finds all reachable pairs of states. We use this idea to extend our previous definition of intersection to cyclic ECTAs: the algorithm tracks all previous visited pairs of nodes, and creates a recursive reference upon seeing the same pair twice.

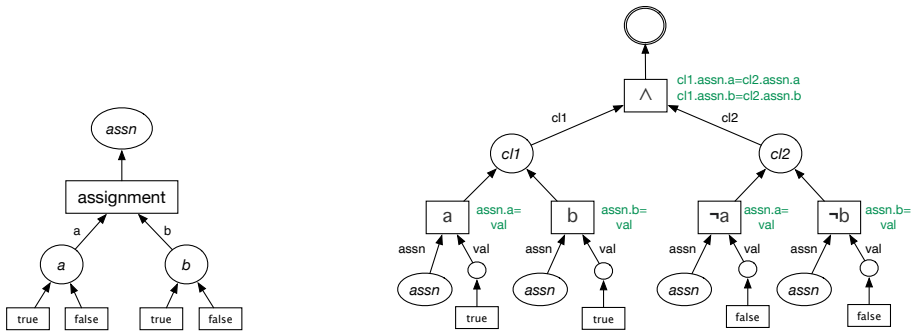
More formally, we define $n_1 \sqcap n_2$ in terms of a helper operation, $n_1 \underset{S}{\sqcap} n_2$ (intersection tracking the set of previously-visited pairs); which in turn invokes the helper $n_1 \underset{S}{\tilde{\sqcap}} n_2$. We assume a function $\mathsf{var}(\{n_1, n_2\})$ mapping an unordered pair of nodes to a unique named variable for that pair. We use the notation $n_1 \in n_2$ to mean that some descendant of n_1 is equal to n_2.

Let $n_1, n_2 \in N$ be two ECTAs, $S \subseteq \binom{N}{2}$ be a set of unordered pairs of ECTAs, and define $S' = S \cup \{\{n_1, n_2\}\}$. Then define:

$$n_1 \underset{S}{\sqcap} n_2 = \begin{cases} \mathsf{var}(n_1, n_2) & \{n_1, n_2\} \in S \\ \mu z.n_1 \underset{S'}{\tilde{\sqcap}} n_2 & z = \mathsf{var}(\{n_1, n_2\}) \wedge z \in (n_1 \underset{S'}{\tilde{\sqcap}} n_2) \\ n_1 \underset{S'}{\tilde{\sqcap}} n_2 & \text{otherwise} \end{cases}$$

The remainder of the definition is almost identical to the definition for acyclic ECTAs, except that recursive nodes are first unfolded. Let $\mathsf{unfold}(n_1) = \mathsf{U}(\overline{e_1})$ and $\mathsf{unfold}(n_2) = \mathsf{U}(\overline{e_2})$. Then:

$$n_1 \underset{S}{\tilde{\sqcap}} n_2 = \mathsf{U}\left(\left\{ e_1^i \underset{S}{\sqcap} e_2^j \;\middle|\; e_1^i \in \overline{e_1}, e_2^j \in \overline{e_2} \right\}\right)$$



(a) Variable assignment (b) CNF formula

Fig. 14. ECTA encoding of a CNF formula $(a \vee b) \wedge (\neg a \vee \neg b)$

Let $e_1 = \mathbf{\Pi}(s_1, [n_1^0 \ldots n_1^{k-1}], C_1)$, $e_2 = \mathbf{\Pi}(s_2, [n_2^0 \ldots n_2^{l-1}], C_2)$ be two transitions, then:

$$e_1 \underset{S}{\sqcap} e_2 = \begin{cases} \mathbf{\Pi}(s_1, [n_1^0 \underset{S}{\sqcap} n_2^0, \ldots, n_1^{k-1} \underset{S}{\sqcap} n_2^{k-1}, C_1 \cup C_2) & \text{if } s_1 = s_2 \text{ and } C_1 \cup C_2 \text{ is consistent} \\ \mathbf{\Pi}_\perp & \text{otherwise} \end{cases}$$

Now, define $n_1 \sqcap n_2 = n_1 \underset{\emptyset}{\sqcap} n_2$.

Static Reduction. Recall from §3.5 that static reduction is defined in terms of intersection at a path, which in turn relies on the definition of nodes at path. Fig. 13 extends these operations to unfold recursive nodes until the ECTA enters lasso form, at least with regards to the PEC under consideration. Then the rest of the static reduction algorithm remains unchanged.

Enumeration. Adapting enumeration to cyclic ECTAs requires a single change: the new Choose-μ rule in Fig. 13 unfolds recursive nodes referenced by some ancestor's constraint. This rule continues unfolding such nodes so long as they are referenced by a parent's constraint, at which point it is a fully enumerated node. Note that a fully-enumerated state will necessarily be in lasso form.

6 IMPLEMENTATION

We have implemented ECTAs in a library called ecta (pronounced as in "nectarine"). ecta is implemented in 3000 lines of Haskell, with an additional 660 lines of tests. ecta has been very carefully optimized, and features heavy memoization based on a mutable hashtable library.

7 APPLICATIONS

This section gives two examples of problem domains that can be reduced to ECTA enumeration: boolean satisfiability (SAT; §7.1) and type-driven program synthesis (§7.2). The second domain has already been introduced informally in §2; here we present its encoding in full generality, and in §8 we evaluate our encoding against a state-of-the-art synthesizer Hoogle+. The purpose of presenting the first domain is to demonstrate the versatility of ECTAs, not to compete with highly-engineered industrial SAT solvers; hence we leave the SAT domain out of empirical evaluation.

7.1 Boolean Satisfiability

Problem Statement. Given a propositional formula in *conjunctive normal form* (CNF), the SAT problem is to find a satisfying assignment to its variables. A CNF formula is a conjunction of *clauses*,

where each clause is a disjunction of *literals*, and each literal is either a variable or its negation. For example, the CNF formula $(a \vee b) \wedge (\neg a \vee \neg b)$ has two satisfying assignments: $\{a, \neg b\}$ and $\{\neg a, b\}$.

Encoding. Fig. 14 illustrates our ECTA encoding for the above formula. The sub-automaton *assn* in Fig. 14a represents the set of all possible variable assignments. The `assignment` transition has one child per variable, and each variable node has two alternatives: `true` and `false`; hence, to extract a term from *assn* one must pick a value for each variable.

The ECTA for the entire CNF formula is shown in Fig. 14b; this ECTA has a single top-level conjunction transition $\wedge$, with one child per clause. Each clause node has one alternative per literal in that clause: the choice between these alternatives corresponds to picking which literal is responsible for making the clause true. Each literal transition—such as `a` or `¬a`—has two children: `assn` is its local copy of the assignment sub-automaton and `val` is the Boolean value that this literal assigns to its variable. The constraint on the literal—such as `assn.a = val`—restricts its local assignment in such a way that the literal evaluates to true. Finally, the constraints on the $\wedge$ transition force all local assignments to coincide. Note that, while the various `assn` nodes are shared in memory, each occurrence of this node is an independent choice unless so constrained. The reader might be wondering why we chose to split these constraints per-variable instead of simply writing `cl1.assn = cl2.assn`; as we explain next, this helps enumeration discover inconsistent assignments quickly.

SAT Solving as ECTA Enumeration. With this encoding, the general-purpose ECTA enumeration algorithm from §4 turns into a SAT solver.[12] Specifically, once Enumerate has found a fully enumerated state, the satisfying assignment can be read off the children of any `assignment` symbol in that state; note that if we let Enumerate run past the first result, it will enumerate all satisfying assignments, modulo irrelevant variables (see §4.4).

The overall solving procedure amounts to choosing a literal from each clause and backtracking whenever the assignment becomes inconsistent. For example, suppose Enumerate has chosen `a` from *cl1*; the enumeration state σ now contains variables v_a and v_b, which store assignments for a and b consistent with the current choices (that is, v_a is restricted to `true`, while v_b still allows both choices). If the algorithm now attempts to make an inconsistent choice of `¬a` from *cl2*, this inconsistency is discovered immediately when `¬a.val` is suspended and intersected with v_a.

7.2 Type-Driven Program Synthesis

Problem Statement. We are interested in the following *type-driven program synthesis* problem:[13] given a type T, called the query type, and a components library Λ, which maps component names to their types, enumerate terms of type T built out of compositions of components from Λ. For example, a Haskell programmer might be interested in a code snippet that, given a list of optional values, finds the first element that is not `Nothing` (and returns a default value if such an element does not exist). The programmer might pose this as a type-driven synthesis problem, where Λ is the Haskell standard library, and the query T is `a → [Maybe a] → a`. Given this problem, the state-of-the-art type-driven synthesizer Hoogle+ [Guo et al. 2020; James et al. 2020] returns a list of candidate programs that includes the desired solution: λ`def mbs → fromMaybe def (listToMaybe (catMaybes mbs))`.

In this section we adopt the setting of Hoogle+, where components can be both *polymorphic* and *higher-order*, both of which make the synthesis problem significantly harder. On the other hand, also following Hoogle+, we do not consider synthesis of inner lambda abstractions: that is, arguments to higher-order functions can be partial applications but not lambdas.

[12] A curious reader might be wondering why don't we go the other direction: encode an ECTA into a SAT formula and use a SAT solver for ECTA enumeration; this is not possible in general, as we discuss in more detail in §10.

[13] This problem is also known as *type inhabitation* [Urzyczyn 1997] and *composition synthesis* [Heineman et al. 2016].

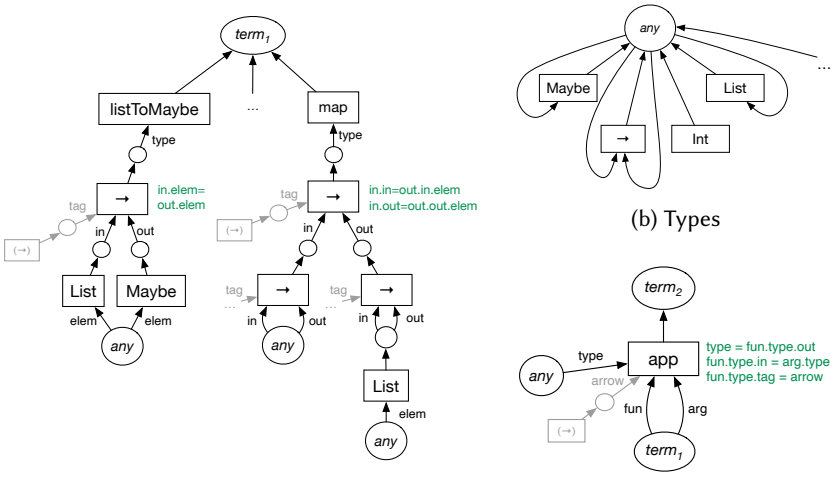


Fig. 15. Encoding of variables, types, and fixed-size terms in Hectare.

Hoogle+ Limitations. Hoogle+ works by encoding a synthesis problem into a data structure called *type-transition net*: a Petri net, where places (nodes) correspond to types, and transitions correspond to components; the synthesis problem then reduces to finding a path from the input types to the output type of the query. This encoding has two major limitations:

(1) *No native support for polymorphic components.* In the presence of polymorphism, the space of types that can appear in a well-typed program becomes infinite. Because types are encoded as places, a finite Petri net cannot represent all candidate programs. Instead, Hoogle+ employs a sophisticated abstraction-refinement loop to build a series of Petri nets that encode increasingly precise approximations of the set of types of interest.

(2) *No native support for higher-order components.* Because components are encoded as transitions with a fixed arity—they transform a fixed number of types into a single type—all components must always be fully applied. This precludes the use of higher-order components: for example, in `foldr (+) 0 xs`, the binary component `(+)` is not fully applied. To circumvent this limitation, Hoogle+ must add a separate *nullary copy* of the `(+)` component to the library. Since these duplicate components bloat the library and slow down synthesis, in practice only a few popular components are duplicated, thereby limiting the practicality of the synthesizer.

In this section we present Hectare (Hoogle+: ECTA REvision), our encoding of type-driven synthesis as ECTA enumeration. This encoding has native support for both polymorphic and higher-order components, without the need for an expensive refinement loop or duplicate components.

Encoding Types. Recall that §2 (Fig. 4) introduced an encoding for a limited form of polymorphism, where the type variable α in a type like $\alpha \to \alpha$ could be instantiated only with base types. We now generalize this encoding so that α can be instantiated with *any type*, with arbitrarily nested applications of type constructors. The infinite space of all types can be finitely encoded as a recursive node *any*, as shown in Fig. 15b. The *any* node has one child per type constructor in Λ, with non-nullary type constructors looping back to *any*. Now the type $\alpha \to \alpha$ can be represented as a $\to$ transition, whose children are both *any* (and are constrained to equal each other).

Encoding Components. The simplified encoding in §2 splits components into different nodes by their arity (*e.g.* the nodes *scalar* and *unary* in Fig. 2); this was necessary given our simplified

encoding of function types, but as we mentioned above, such arity-specific encoding precludes partial applications. Fig. 15a illustrates the generalized encoding of components in HECTARE. Here all components, regardless of arity, are gathered in single node $term_1$ ("terms of size one"). Each component is annotated with its `type`; function types are represented using the $\rightarrow$ transition with two child types, `in` and `out` (for now, ignore the grayed out edges labeled `tag`, we explain those below). Fig. 15a showcases the type encoding for two polymorphic components: `listToMaybe` :: $[\alpha] \rightarrow$ `Maybe` α and `map` :: $(\alpha \rightarrow \beta) \rightarrow [\alpha] \rightarrow [\beta]$; as before, all occurrences of the same type variable are related by equality constraints, shown in green.

Encoding Applications. Fig. 15c illustrates the HECTARE encoding of size-two terms. As before, the application transition `app` has two children `fun` and `arg`, but now they are both represented by the same node $term_1$; hence this encoding supports partial applications, such as `map listToMaybe`.

We now explain the purpose of the grayed-out parts of Fig. 15. The `app` node must ensure that its `fun` child has an arrow type. The mere presence of `fun.type.in` and `fun.type.out` in its first two constraints does not suffice: recall that the actual ECTA library refers to children by index instead of by name, and hence any other binary type constructor (such as `Either`) could satisfy those two constraints. This would lead to accepting ill-typed programs, such as `Left` x y. To circumvent this issue, we introduce a special tag transition ($\rightarrow$), which occurs nowhere else but as a first child of every $\rightarrow$ transition; by constraining the first child of `fun` to be ($\rightarrow$), `app` effectively ensures that it is indeed a function (see the last constraint on `app`).[14]

This encoding of application terms generalizes from size-two terms to terms of arbitrary fixed size n as follows: the node $term_n$ has $n-1$ incoming `app` transitions, where the i-th transition ($i \in 1..n-1$) has children $term_i$ and $term_{n-i}$.

Synthesis Algorithm. So far we have discussed how to encode the space of all well-typed terms of size n. Let us now proceed to the top-level synthesis algorithm of HECTARE. Given a query type, such as `a` $\rightarrow$ [`Maybe a`] $\rightarrow$ `a`, HECTARE first adds the inputs of the query (here `def` :: `a` and `mbs` :: [`Maybe a`]) to the node $term_1$, as if they were components. The algorithm then iterates over program sizes $n \geq 1$; for each size n, it constructs the ECTA $term_n$ and restricts its top-level type to the return type of the query (here `a`), following the recipe illustrated in Fig. 3. The algorithm then statically reduces all constraints in the restricted ECTA and enumerates all terms accepted by the resulting reduced ECTA, before moving on to the next size n. Note that the type variables *of the query* (here `a`) are represented as type constructors and not as the *any* node, since those type variables are universally quantified.

Enforcing Relevancy. Existing type-driven synthesizers [Feng et al. 2017; Guo et al. 2020] restrict synthesis results to *relevantly typed* terms—that is, terms that use all the inputs of the query. Without such relevancy restriction, any synthesis algorithm gets bogged down by short but meaningless programs. HECTARE enforces relevancy via a slight modification to the simple synthesis algorithm outlined above: it splits every $term_n$ node into 2^k nodes, where k is the number of inputs in the query. In our example, there are four nodes at each term size: $term_n^{\{\texttt{def},\texttt{mbs}\}}$, $term_n^{\{\texttt{def}\}}$, $term_n^{\{\texttt{mbs}\}}$, $term_n^{\emptyset}$, each representing terms that must mention the corresponding set of inputs. When constructing a new term node, say $term_2^{\{\texttt{def}\}}$, HECTARE considers all applications of $term_1^P$ to $term_1^Q$ such that $P \cup Q = \{\texttt{def}\}$. At the top level, only $term_n^{\{\texttt{def},\texttt{mbs}\}}$ is connected to the accepting node. Although the

[14] Stepping back, tags are required in this encoding because the space of types in the HECTARE ECTA is a sum of two distinct variants: `Type` ::= ($\rightarrow$)(`Type`, `Type`) | c(`Type`$*$). The tags exist to discriminate between the $\rightarrow$ variant and the variant c(`Type`, `Type`), where c is any other binary type constructor, such as `Either`. One might ask: why not build the ability to discriminate between variants of a sum directly into the ECTA? One way to do this is by referencing children by name instead of by index, as in §2. This is a viable alternative approach, but it is less efficient: an implementation based on names needs to compare them at *every access*, whereas one based on indices only needs to do so at sites where confusion is possible.

Table 1. Three sample queries and corresponding solutions from two benchmark suites.

Suite	Name	Query	Expected solution
Hoogle+	mergeEither	`Either a (Either a b) → Either a b`	`λe → either Left id e`
	headLast	`[a] → (a, a)`	`λxs → (head xs, last xs)`
	both	`(a → b) → (a, a) → (b, b)`	`λf p → (f (fst p), f (snd p))`
Stack-Overflow	multiIndex	`[a] → [Int] → [a]`	`λxs is → map ((!!) xs) is`
	splitOn	`Eq a ⇒ a → [a] → [[a]]`	`λx xs → groupBy (on (&&) (/= x)) xs`
	matchedKeys	`(b → Bool) → [(a, b)] → [a]`	`λp xs → map fst (filter (p . snd) xs)`

number of term-nodes in this encoding grows exponentially with the number of inputs, this is not a problem in practice, since the number of inputs is typically small; note also that due to hash consing in ECTA, the overlapping component sets are not actually duplicated.

8 EVALUATION

As we explained in §7.2, we used the ECTA library to implement HECTARE, a type-driven component-based synthesizer for Haskell. In this section, we evaluate the performance of HECTARE and compare it with the state-of-the-art synthesizer HOOGLE+, based on an SMT encoding of Petri-net reachability [Guo et al. 2020]. Both tools are written in Haskell, but the HOOGLE+ implementation (excluding tests and parsing) contains a whopping 4000 LOC, while HECTARE only contains 400. Although code size is an imperfect measure of development effort, these numbers suggest that the ECTA library has the potential to significantly simplify the development of program synthesizers.

We designed our evaluation to answer the following research questions:

(RQ1) How does HECTARE compare against HOOGLE+ on existing and new benchmarks?
(RQ2) How significant are the benefits of static and dynamic reduction in program synthesis?

All experiments were conducted on an Intel Core i9-10850K CPU with 32 GB memory.

8.1 Comparison on HOOGLE+ Benchmarks

Experiment Setup. For our main experiment, we compare the two synthesizers on the benchmark suite from the latest HOOGLE+ publication [James et al. 2020]. This suite includes 45 synthesis queries, and a library of 291 components from 12 popular Haskell modules. These benchmarks are non-trivial: the expected solutions range in size from 3 to 9, with the average size of 4.7; 40% of the components are polymorphic, and 44% of the queries require using a higher-order component. Three sample queries from this suite are listed at the top of Tab. 1. The solutions to these queries have sizes 4, 5, and 7 respectively, and `mergeEither` uses a higher-order component `either`.

Both HOOGLE+ and HECTARE yield candidate programs one at a time, gradually increasing the size of the programs they consider. For both tools, our test harness terminates the search once the expected solution has been found (or the timeout of 300 seconds has been reached). We report the average time to expected solution over three runs. We configured HECTARE to perform static reduction on all constraints prior to running the fast enumeration procedure of §4.3, repeating this operation up to 30 rounds or until the automaton converges.

Results. Fig. 16 plots the number of benchmarks solved *vs.* time for both synthesizers. Within the timeout, HECTARE solves 43 out of 45 benchmarks, whereas HOOGLE+ only solves 39. Importantly, as we show in Fig. 17, on commonly solved benchmarks HECTARE is significantly faster: it achieves an average speedup (geometric mean) of 7× on this suite, solving all but two tasks faster than HOOGLE+.

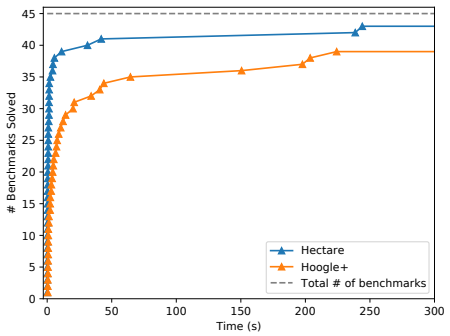


Fig. 16. Benchmarks solved vs time for Hectare and Hoogle+ on Hoogle+ benchmarks.

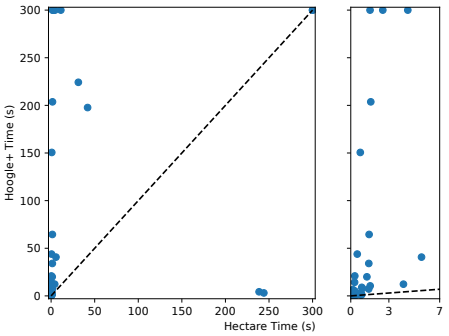


Fig. 17. Synthesis times of Hectare against Hoogle+ on Hoogle+ benchmarks.

Fast synthesis times are especially important if a synthesizer is to be used interactively. As shown in the zoomed-in scatter plot in Fig. 17 (right), Hectare also vastly outperforms Hoogle+ if we consider a shorter timeout of seven seconds, commonly used for interactive synthesizers [Ferdowsifard et al. 2021]; in fact, Hectare solves 84% of the benchmarks within seven seconds.

The poor performance of Hoogle+ can be mainly attributed to the brittleness of the abstraction-refinement loop it uses to support polymorphic components (§7.2). For example, the `headLast` benchmark from Tab. 1 is one of the queries where Hoogle+ times out, while Hectare only takes 2.5 seconds. Upon closer inspection, Hoogle+ is unable to create an accurate type abstraction for this query and ends up wasting a lot of time enumerating ill-typed terms. Hectare, in contrast, natively supports polymorphic components via recursive nodes, which leads to more predictable performance. On the other hand, the two benchmarks where Hectare is slower than Hoogle+ both involve deconstructing a `Pair` and using both of its fields (`both` from Tab. 1 is one of these benchmarks). Hoogle+ solves these queries using a special treatment of `Pairs`: it introduces a single component that projects both fields of a pair simultaneously, which makes the solutions to these queries much shorter; in Hectare, we did not find a straightforward way to add this trick.

In general, we conclude that *Hectare is effective in solving type-driven synthesis tasks and outperforms a state-of-the-art tool on 89% of their benchmarks with 7× speedup on average.*

8.2 Comparison on StackOverflow Benchmarks

Benchmark Selection. Recall that another limitation of Hoogle+ we discussed in §7.2 is its restricted support for higher-order functions. In fact, the original Hoogle+ configuration contains only *nine* components whose nullary versions are added to the Petri net (and which consequently can appear in arguments to higher-order functions). In order to push the limits of both tools and demonstrate the benefits of Hectare's native encoding, we assembled an additional benchmark suite focusing on higher-order functions. To this end, we searched StackOverflow for Haskell programming questions; for each question, we attempted to construct an expected solution using only applications of library components; we excluded tasks that can be solved without higher-order functions or require unsupported features (such as higher-kinded type variables and inner lambda abstractions). This left us with 19 synthesis queries. The new benchmark suite is generally more complex than the original Hoogle+ suite: expected solutions range in size from 4 to 9, with the average of 6.2; all of these programs include partial applications as arguments to higher-order components. Three sample queries are shown at the bottom of Tab. 1.

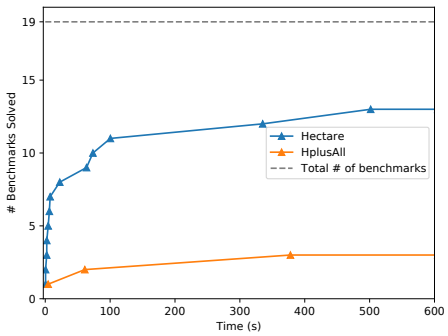


Fig. 18. Comparison of synthesis time on higher-order benchmarks between Hectare and HplusAll.

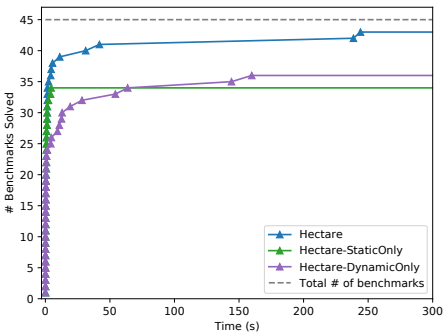


Fig. 19. Comparison of synthesis performance between Hectare and its two variants.

Experiment Setup. To run the newly collected benchmarks, we augmented the original component set from Hoogle+ with seven components required in these benchmarks. We also created a variant of Hoogle+ called HplusAll, in which we added nullary copies of all components into the Petri net (HplusAll thus has the same expressiveness as Hectare). As before, we record the time to expected solution, repeat the measurement three times, and report the average time; to accommodate the increased benchmark complexity, we use a longer timeout of 600 seconds.

Results. Unsurprisingly, the original Hoogle+ cannot solve any of the new benchmarks: most of them require using new components in higher-order arguments (and the rest are simply too large). The results for HplusAll and Hectare are shown in Fig. 18. Although the search space of HplusAll does include all the new benchmarks, it still fares poorly, solving only 3 out of 19. The reason is that adding nullary versions of all components blows up the Petri net and makes the reachability problem intractable. In contrast, Hectare's native support for partial applications enables it to solve 13 out of 19 tasks in this challenging suite, achieving 40× speedup on the three commonly solved benchmarks. We therefore conclude that *the benefits of ECTA-based synthesis are even more pronounced on larger benchmarks focused on higher-order functions.*

8.3 Benefits of Static and Dynamic Reduction

Experiment Setup. To isolate the contributions of static and dynamic reduction, we compare Hectare with its three variants: Hectare-StaticOnly, Hectare-DynamicOnly, and Hectare-Naïve, which forgo one or both kinds of reduction, respectively. Specifically, both Hectare-StaticOnly and Hectare-Naïve, use a naïve "rejection-sampling" enumeration. Note that in the presence of recursive nodes, such as the *any* type, the naïve enumeration tends to get "stuck", constructing infinitely many spurious terms and never finding one that satisfies the constraints. To prevent this behavior, we limit the unfolding depth of recursive nodes to three, which is sufficient to solve all the benchmarks. We run the three variants on the Hoogle+ benchmarks with a timeout of 300 seconds and report the average time to expected solution over three runs.

Results. Fig. 19 plots the number of benchmarks solved *vs.* time for Hectare and its variants. Hectare-Naïve is omitted from the plot because it cannot solve *any benchmarks*: it spends most of its time unfolding the recursive *any* node, or in other words, blindly going through all possible instantiations of every polymorphic component. The other two variants fare significantly better: Hectare-StaticOnly and Hectare-DynamicOnly are able to solve 34 and 36 tasks, respectively.

That said, as the tasks get harder, Hectare still outperforms these variants drastically: in particular, the variants cannot solve any benchmarks of size six or larger.

A closer look at Fig. 19 reveals a curious difference: Hectare-StaticOnly is "all-or-nothing": it performs as well as Hectare on easy benchmarks, but then completely falls flat; Hectare-DynamicOnly, in contrast, demonstrates a more gradual degradation of performance. To understand why, recall that the biggest time sink during enumeration is blindly unfolding the recursive *any* nodes. Static reduction can sometimes get rid of *any* nodes entirely, making the resulting ECTA small enough that any enumeration algorithm would do; when it fails to do so, however, naïve enumeration spends all its time in *any* nodes. Dynamic reduction, on the other hand, provides a more gradual yet robust approach to dealing with *any* nodes, via Suspend. In summary, we find that *both static and dynamic reduction individually are critical to the performance of ECTA-based programs synthesis, and moreover, they complement each other's strengths.*

9 RELATED WORK

Constrained Tree Automata. Tree automata have long been used to represent sets of terms in term rewriting [Dauchet 1993; Feuillade et al. 2004; Geser et al. 2007], and we are not the first to consider adding equality constraints to handle nonlinear rewrites. In fact, in 1995, Dauchet introduced a data structure very similar to our ECTAs, called *reduction automata* [Dauchet et al. 1995]. In fact, reduction automata are more expressive than ECTAs, as they also allow *disequality constraints*, including disequalities (but not equalities) on cycles. Unfortunately, allowing disequalities—or other classes of constraints for that matter—precludes efficient static and dynamic reduction based on automata intersection. For that reason, we consider ECTAs to be a sweet spot: expressive enough to encode a variety of interesting problems, yet restricted enough to enable fast enumeration.

Other prior work on constrained tree automata [Barguñó et al. 2010; Barguñó et al. 2013; Bogaert et al. 1999; Bogaert and Tison 1992; Reuß and Seidl 2010] similarly focuses on theoretical aspects, such as worst-case complexity and decidability results, and we have found no reference to these data structures being used in a practical system in the 30 years since their introduction.

Attribute grammars [Knuth 1968; Paakki 1995; Van Wyk et al. 2010] augment context-free grammars with a number of equations of the form ⟨attribute⟩ = ⟨expression⟩. This notation resembles a constraint system over trees, but those equations are actually unidirectional assignments; attribute grammars compute values over trues, but do not constrain them.

Unconstrainted FTAs, VSAs, and E-Graphs. In contrast to the purely theoretical work on constrained tree automata, their unconstrained counterparts, as well as VSAs and e-graphs, have enjoyed practical applications in program synthesis [Gulwani 2011; Nandi et al. 2020, 2021; Polozov and Gulwani 2015; Wang et al. 2017, 2018; Willsey et al. 2021] and related areas, such as theorem proving [Detlefs et al. 2005], superoptimization [Yang et al. 2021], and semantic code search [Premtoon et al. 2020]. One important feature of these data structures, which ECTAs currently lack, is the ability extract an optimal term according to a user-defined cost function. It is not surprising that ECTAs have a slightly different focus, since in the presence of constraints extracting terms regardless of cost becomes hard—at least as hard as SAT solving. Extracting optimal terms would be akin to MaxSAT solving [Krentel 1986]; we leave this non-trivial extension to future work.

Finally note that unlike FTAs and VSAs, e-graphs are used to represent a *congruence relation* over terms, as opposed to an arbitrary term space; hence adding equality constraints to an e-graph is less meaningful. Returning to our introductory example in Fig. 1, an e-graph equivalent to the FTA in Fig. 1b would actually encode that a, b, and c, are all *equivalent* to each other; hence it is hard to imagine why one would want to represent only the terms of the form $+(f(X), f(X))$ but not $+(f(X), f(Y))$, because all these terms are equivalent.

10 CONCLUSIONS AND FUTURE WORK

This paper has introduced *equality-constrained tree automata* (ECTAs) and contributed an efficient implementation of this new data structure in the ECTA library. We think of ECTAs as a general-purpose language for expressing constraints over terms, and the ECTA library as a solver for these constraints. Although in this paper we only discussed two concrete examples of properties that can be encoded with ECTAs—boolean satisfiability and well-typing—in the future we hope to see many fruitful applications in a wide range of domains.

***ECTA* vs. *SMT*.** Instead of developing a custom solver for ECTAs, wouldn't it be better to simply translate ECTAs into SAT or SMT constraints, and use existing, well-engineered solvers? A natural idea is to introduce a variable per ECTA node, whose value represents the choice of incoming transition, and to translate ECTA constraints into equalities between these variables. This simple idea, however, does not work: because the choice is made independently every time a node is visited, this encoding would require unfolding the ECTA *into a tree*. This is a complete non-starter for cyclic ECTAs (like the HECTARE encoding of §7.2), since the corresponding tree is infinite. For acyclic ECTAs, the tree is finite but might be exponential in the size of the ECTA (since we need to "un-share" all the shared paths in the DAG).

More generally, the problem of finding an ECTA inhabitant is not in NP, because the smallest tree represented by an ECTA can be exponential in the size of the ECTA (as we illustrated in Fig. 10); hence a general and efficient SAT encoding is not possible. Although future work might develop a clever SMT encoding using advanced theories, we believe this problem is far from trivial. After all, HOOGLE+ uses an SMT encoding that is specifically tailored to the type inhabitation problem (*i.e.* it is less general than ECTA), and yet it is less efficient. As we discussed in §7.2, the main source of this inefficiency is polymorphism, which makes the search space of types *infinite* and precludes a "one-shot" SMT encoding, requiring HOOGLE+ to go through a series of finite approximations of the space of types to consider. Instead, a cyclic ECTA is able to represent the entire infinite space of types at once, and ECTA enumeration is able to explore this space efficiently, as our experiments show. We anticipate that ECTAs will outperform SMT solvers on other similar problems that require searching an *infinite yet constrained space of terms*.

Future Work. One avenue for extension is to enrich the constraint language supported by ECTAs. The key ingredient for efficiency is that there exists a constraint-propagation mechanism that can be interleaved with CHOOSE. Intersection is this constraint-propagation mechanism for equality, but there may be others. For example, disequality constraints could be processed by creating an alternative rule to SUSPEND which tracks both sides of a disequality, and modifying CHOOSE to discharge disequality constraints or propagate them into subterms as symbols are selected.

Another path for extension is to relax the requirement for no constraints on cycles. A careful reader may notice that Theorem 5.1 only impedes emptiness-checking; enumerating all satisfying terms up to a fixed size is trivially decidable. Currently HECTARE creates many ECTA nodes for different term sizes, using a meta-program to iterate through successive ECTAs. With constraints on cycles, this meta-program could be internalized, further shortening the HECTARE implementation.

ACKNOWLEDGMENTS

Thank you to Joshua Pollock for discovering and informing us of the connection between e-graphs and tree automata. Without this initial insight, our discoveries in this paper may have never occurred. This work has been supported by the US Air Force, AFRL/RIKE, DARPA under Contract No. FA8750-20-C-0208, and the National Science Foundation under Grant No. 1943623. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the US Air Force, AFRL/RIKE, DARPA or NSF.

REFERENCES

Michael D Adams and Matthew Might. 2017. Restricting Grammars with Tree Automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–25. https://doi.org/10.1145/3133906

Leo Bachmair and Nachum Dershowitz. 1994. Equational Inference, Canonical Proofs, and Proof Orderings. *Journal of the ACM (JACM)* 41, 2 (1994), 236–276. https://doi.org/10.1145/174652.174655

Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. 2010. The Emptiness Problem for Tree Automata with Global Constraints. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*. IEEE, 263–272. https://doi.org/10.1109/LICS.2010.28

Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. 2013. Decidable Classes of Tree Automata Mixing Local and Global Constraints Modulo Flat Theories. *Logical Methods in Computer Science* 9 (02 2013). https://doi.org/10.2168/LMCS-9(2:1)2013

Bruno Bogaert, Franck Seynhaeve, and Sophie Tison. 1999. The Recognizability Problem for Tree Automata with Comparisons Between Brothers. In *International Conference on Foundations of Software Science and Computation Structure*. Springer, 150–164. https://doi.org/10.1007/3-540-49019-1_11

Bruno Bogaert and Sophie Tison. 1992. Equality and Disequality Constraints on Direct Subterms in Tree Automata. In *Annual Symposium on Theoretical Aspects of Computer Science*. Springer, 159–171. https://doi.org/10.1007/3-540-55210-3_181

Max Dauchet. 1993. Rewriting and Tree Automata. In *French School on Theoretical Computer Science*. Springer, 95–113. https://doi.org/10.1007/3-540-59340-3_8

Max Dauchet, Anne-Cécile Caron, and Jean-Luc Coquidé. 1995. Automata for Reduction Properties Solving. *Journal of Symbolic Computation* 20, 2 (1995), 215–233. https://doi.org/10.1006/jsco.1995.1048

David Detlefs, Greg Nelson, and James B Saxe. 2005. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM (JACM)* 52, 3 (2005), 365–473. https://doi.org/10.1145/1066100.1066102

Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-based synthesis for complex APIs. In *POPL*. https://doi.org/10.1145/3009837.3009851

Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. 2021. LooPy: Interactive Program Synthesis with Control Structures. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 153 (oct 2021), 29 pages. https://doi.org/10.1145/3485530

Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. 2004. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasoning* 33, 3 (2004), 341–383. https://doi.org/10.1007/s10817-004-6246-0

Alfons Geser, Dieter Hofbauer, Johannes Waldmann, and Hans Zantema. 2007. On Tree Automata that Certify Termination of Left-Linear Term Rewriting Systems. *Information and Computation* 205, 4 (2007), 512–534. https://doi.org/10.1007/978-3-540-32033-3_26

Matthías Páll Gissurarson. 2018. Suggesting Valid Hole Fits for Typed-Holes (Experience Report). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell* (St. Louis, MO, USA) *(Haskell 2018)*. Association for Computing Machinery, 179–185. https://doi.org/10.1145/3299711.3242760

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets using Input-Output Examples. *ACM Sigplan Notices* 46, 1 (2011), 317–330. https://doi.org/10.1145/1926385.1926423

Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2020. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.* 4, POPL (2020), 12:1–12:28. https://doi.org/10.1145/3371080

George T. Heineman, Jan Bessai, Boris Düdder, and Jakob Rehof. 2016. A Long and Winding Road Towards Modular Synthesis. In *Leveraging Applications of Formal Methods, Verification and Validation: Foundational Techniques - 7th International Symposium, ISoLA 2016, Imperial, Corfu, Greece, October 10-14, 2016, Proceedings, Part I*. 303–317. https://doi.org/10.1007/978-3-319-47166-2_21

Michael B James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. Digging for Fold: Synthesis-Aided API Discovery for Haskell. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–27. https://doi.org/10.1145/3428273

Donald E Knuth. 1968. Semantics of Context-Free Languages. *Mathematical Systems Theory* 2, 2 (1968), 127–145. https://doi.org/10.1007/BF01692511

James Koppel. 2021. Version Space Algebras are Acyclic Tree Automata. https://doi.org/10.48550/arXiv.2107.12568 arXiv:2107.12568 [cs.PL]

James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces (Extended Version). https://doi.org/10.48550/ARXIV.2206.07828

M W Krentel. 1986. The Complexity of Optimization Problems. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing* (Berkeley, California, USA) *(STOC '86)*. Association for Computing Machinery, New York, NY, USA, 69–76. https://doi.org/10.1145/12130.12138

Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. 2003. Programming by Demonstration Using Version Space Algebra. *Machine Learning* 53, 1 (2003), 111–156. https://doi.org/10.1023/A:1025671410623

Neil Mitchell. 2004. Hoogle. https://www.haskell.org/hoogle/.

Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. 2020. Synthesizing Structured CAD models with Equality Saturation and Inverse Transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 31–44. https://doi.org/10.1145/3385412.3386012

Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. 2021. Rewrite Rule Inference Using Equality Saturation. *Proc. ACM Program. Lang.* 5, OOPSLA, Article 119 (oct 2021), 28 pages. https://doi.org/10.1145/3485496

Greg Nelson and Derek C. Oppen. 1980. Fast Decision Procedures Based on Congruence Closure. *J. ACM* 27, 2 (1980), 356–364. https://doi.org/10.1145/322186.322198

Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL (T). *Journal of the ACM (JACM)* 53, 6 (2006), 937–977. https://doi.org/10.1145/1217856.1217859

Jukka Paakki. 1995. Attribute Grammar Paradigms—A High-Level Methodology in Language Implementation. *ACM Computing Surveys (CSUR)* 27, 2 (1995), 196–255. https://doi.org/10.1145/210376.197409

Joshua Pollock and Altan Haan. 2021. E-Graphs Are Minimal Deterministic Finite Tree Automata (DFTAs) · Discussion #104 · egraphs-good/egg. https://github.com/egraphs-good/egg/discussions/104

Oleksandr Polozov and Sumit Gulwani. 2015. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 107–126. https://doi.org/10.1145/2858965.2814310

Varot Premtoon, James Koppel, and Armando Solar-Lezama. 2020. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1066–1082. https://doi.org/10.1145/3385412.3386001

Andreas Reuß and Helmut Seidl. 2010. Bottom-up Tree Automata with Term Constraints. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*. Springer, 581–593. https://doi.org/10.1007/978-3-642-16242-8_41

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 264–276. https://doi.org/10.1145/1480881.1480915

Pawel Urzyczyn. 1997. Inhabitation in Typed Lambda-Calculi (A Syntactic Approach). In *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*. 373–389. https://doi.org/10.1007/3-540-62688-3_47

Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. 2010. Silver: An Extensible Attribute Grammar System. *Science of Computer Programming* 75, 1-2 (2010), 39–54. https://doi.org/10.1016/j.scico.2009.07.004

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2017. Synthesis of Data Completion Scripts using Finite Tree Automata. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–26. https://doi.org/10.1145/3133886

Xinyu Wang, Isil Dillig, and Rishabh Singh. 2018. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.* 2, POPL (2018), 63:1–63:30. https://doi.org/10.1145/3158151

Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. 2021. Egg: Fast and Extensible Equality Saturation. *Proceedings of the ACM on Programming Languages* 5, POPL (2021), 1–29. https://doi.org/10.1145/3434304

Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. 2021. Equality Saturation for Tensor Graph Superoptimization. In *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica (Eds.), Vol. 3. 255–268. https://proceedings.mlsys.org/paper/2021/file/65ded5353c5ee48d0b7d48c591b8f430-Paper.pdf