UNIVERSITY OF CALIFORNIA SAN DIEGO

Type-Directed Program Synthesis for Complex APIs

A dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy

in

Computer Science

by

Zheng Guo

Committee in charge:

Professor Nadia Polikarpova, Chair
Professor Taylor Berg-Kirkpatrick
Professor Judith Ellen Fan
Professor Ranjit Jhala

2024

University of California San Diego

2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First and foremost, I extend my deepest gratitude to my advisor, Nadia Polikarpova, for her invaluable guidance, patience, and support throughout this challenging PhD journey. Her insights and words of encouragement have been a great source of strength to me.

I would also like to thank my dissertation committee: Ranjit Jhala, Judith Ellen Fan, and Taylor Berg-Kirkpatrick, for their time, insightful feedback, and expertise. Their perspectives and critiques were crucial in shaping the research, making this work as comprehensive as it is.

I feel lucky to collaborate with many wonderful researchers during my PhD: Michael B. James, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, David Justo, Jiaxiao Zhou, Ziteng Wang, Shivani Doshi, Hila Peleg, James Koppel, Edsko de Vries, and Armando Solar-Lezama. Working alongside such talented researchers has been an inspiring experience.

I must express my profound gratitude to my family for providing me with unwavering support and continuous encouragement throughout my years of study. This accomplishment would not have been possible without them.

Chapter 1, in part, is a reprint of the material as it appears in Proceedings of the ACM on Programming Languages. Volume 4 (POPL), Article 12. Zheng Guo, Michael B. James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, Nadia Polikarpova. ACM 2020. The dissertation author was a principal author and investigator of this paper.

Chapter 2, in part, is a reprint of the material as it appears in Proceedings of the ACM on Programming Languages. Volume 6 (ICFP), Article 91. James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, Nadia Polikarpova. ACM 2022. The dissertation author was a principal author and investigator of this paper.

Chapter 3, in part, is a reprint of the material as it appears in Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, Nadia Polikarpova. ACM 2022. The dissertation author was a principal author and investigator of this paper.

VITA

2013	Bachelor of Engineering, Shanghai Jiao Tong University
2019	Master of Science, University of California San Diego
2024	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, Nadia Polikarpova. "Searching entangled program spaces". Proceedings of the ACM on Programming Languages. Volume 6 (ICFP), Article 91. August 2022.

Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, Nadia Polikarpova. "Type-directed program synthesis for RESTful APIs". Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2022.

Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, Nadia Polikarpova. "Digging for Fold: Synthesis-Aided API Discovery for Haskell". Proceedings of the ACM on Programming Languages. Volume 4 (OOPSLA). November 2020.

Zheng Guo, Michael B. James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, Nadia Polikarpova. "Program Synthesis by Type-Guided Abstraction Refinement". Proceedings of the ACM on Programming Languages. Volume 4 (POPL), Article 12. January 2020.

ABSTRACT OF THE DISSERTATION

Type-Directed Program Synthesis for Complex APIs

by

Zheng Guo

Doctor of Philosophy in Computer Science

University of California San Diego, 2024

Professor Nadia Polikarpova, Chair

As software systems grow sophisticated, complex APIs are widely used to accelerate program development. However, effectively using these APIs can be challenging due to poor documentation that fails to sufficiently explain API behaviors. This issue highlights the necessity for automating the combination of APIs into programs that satisfy given high-level specifications, thereby reducing the burden on engineers.

This dissertation proposes to use type-directed program synthesis as a method for automating API navigation. In this process, the synthesizer must systematically search through APIs to construct programs that adhere strictly to the provided specifications. This task presents two primary challenges: the effective representation of these specifications and the efficient propagation of relevant information during search. To overcome these challenges, type-directed synthesis leverages compact, type-represented search spaces that enhance the propagation of type information throughout the search process. Although traditional type systems facilitate compact search space representation, they are often too coarse-grained to express user intent precisely. To address this, a semantic type system is introduced to express more advanced constraints on function signatures, effectively ruling out programs that deviate from conventional human-written code.

This dissertation provides a comprehensive analysis of type-directed program synthesis, emphasizing its advantages in handling complex API-driven environments. Through theoretical frameworks and empirical studies, it shows that type-directed synthesis techniques can solve real-world programming tasks within a reasonable time limit. The experimental results demonstrate that the developed synthesis tools significantly improve the feasibility and accuracy of synthesized programs, contributing to more efficient and reliable software development.

Introduction

Application programming interfaces (APIs) have become crucial components of modern software systems due to their ability to manage the increasing complexity of software development. For example, Amazon Web Services offers over two hundred products and services, each with tens or hundreds of API methods. APIs boost software development by providing a library of encapsulated implementations for common functionalities, enabling developers to reuse code across different scenarios. They also promote the modular design of software systems, which facilitates the interchangeability of APIs and allows developers to seamlessly transition between API providers.

As APIs evolve, the number of available methods grows drastically, making it increasingly challenging for developers to quickly grasp the usability of APIs, especially when comprehensive documentation is lacking. This presents engineers with the daunting task of API discovery: given a library of methods, how can they efficiently determine which methods to call and how to call them to achieve their desired goals?

Traditional approaches [14, 66] address this challenge with intelligent code suggestion tools. Typically, these tools first learns statistical information on API or code snippet frequencies from web code corpra or local repositories. Suggestions are then displayed to users in the order of their scores computed from the statistical data and the context. Therefore, these tools can only produce function names or code snippets that appeared in their training corpus, and they are unable to invent unseen expressions during runtime.

Recent research proposes a more advanced technique called *component-based program synthesis* [54], where the algorithm automatically combines methods in the given library and finds programs that satisfy the given specification. This technique has been used to assist programmers to navigate APIs in Java [27] and Scala [47]. The input to a component-based synthesizer can be a type signature or several input-output examples, depending on which information is easier for users to obtain and propagate during the search process.

This dissertation explores the potential of *types* as input for component-based synthesizers. Prior work [27, 47] has shown the capability of types in component-based synthesis. They build type transition graphs (TTGs) for all component, where each component represents a transition from input types to the output type. Subsequently, they reduce the synthesis problem into a graph reachability problem, as each path in the type transition graph corresponds to a set of programs. However, this graph-based search space representation is limited to components with *monomorphic types*. In practical scenarios, component types are often more diverse. In this dissertation, we extend prior work to support types that are more general than monomorphic types. The main contributions are summarized as follows:

Abstract Types and Type-Guided Abstraction Refinement. (chapter 1) Polymorphic datatypes such as `List`, `Map`, etc. are ubiquitous in API methods, particularly in functional programming languages. These types contain type variables, allowing them to be instantiated into an infinite number of concrete types. This leads to an explosion in the number of transitions in TTGs. We introduce *abstract types* that *overapproximate* the infinite set of types. Then an abstract TTG is built and the same graph reachability algorithm is applied to search for programs. Due to the overapproximation, the reachable path may not correspond to well-typed programs in the concrete type system. To overcome this issue, we propose *type-guided abstraction refinement* (TYGAR) to refine the type abstraction and rule out such *spurious programs*. Experimental results demonstrate that this algorithm successfully finds well-typed programs for 98% of collect benchmarks within an average of 1.4 seconds.

Equality-Constrained Tree Automata. (chapter 2) The previously mentioned type transition graph lacks the ability to support polymorphic types without abstraction, but type-guided abstraction refinement is costly as the abstract type transition graph must be reconstructed after each refinement step. Alternatively, we propose a novel data structure called equality-constrained tree automata (ECTA) to compactly represent connections between components with polymorphic types. ECTAs are tree automata annotated with equality constraints, and support recursive nodes that allows representation of type variables in polymorphic types. Additionally, we introduce a fast enumeration algorithm that efficiently extracts terms from the search space, corresponding to the program synthesis step. The experiments show that with ECTAs, the synthesizer solves tasks 7$\times$ faster than TYGAR, while the implementation is only a tenth of the size.

Semantic Types for Program Synthesis. (chapter 3) Although types may not always be native properties for components in all situations, they can serve as good summaries of component behaviors if the type system has the appropriate granularity. To verify this idea, we focus on REST APIs as a case study. In the domain of REST APIs, we develop a semantic type system from their OpenAPI specification, and automatically infer semantic types from execution traces of these APIs. With semantic types in hand, we apply the component-based synthesis algorithm from prior work [42] to generate programs. Our experiments prove that semantic types are helpful and effective in solving real-world benchmarks, successfully finding the desired solution for 23 out of 29 benchmarks collected from online resources.

Chapter 1

Program Synthesis By Type Guided Abstraction Refinement

1.1 Introduction

Consider the task of implementing a function `firstJust def mbs`, which extracts the first non-empty value from a list of options `mbs`, and if none exists, returns a default value `def`. Rather than writing a recursive function, you suspect you can implement it more concisely and idiomatically using components from a standard library. If you are a Haskell programmer, at this point you will likely fire up Hoogle [68], the Haskell's API search engine, and query it with the intended type of `firstJust`, *i.e.* `a → [Maybe a] → a`. The search results will be disappointing, however, since no single API function matches this type[1]. In fact, to implement `firstJust` you need a snippet that composes three library functions from the standard `Data.Maybe` library, like so: `\def mbs → fromMaybe def (listToMaybe (catMaybes mbs))`. Wouldn't you like a tool that could automatically synthesize such snippets from type queries?

Scalable Synthesis via Graph Reachability. In general, our problem of type-directed *component-based synthesis*, reduces to that of finding inhabitants for a given query type [98]. Consequently, one approach is to develop synthesizers based on proof search in intuitionistic logics [5]. However, search becomes intractable in the presence of libraries with hundreds or thousands of components. Several papers address the issue of scalability by rephrasing the problem as one of reachability

[1]We tested this query at the time of writing with the default Hoogle configuration (Hoogle 4).

in a *type transition network* (TTN), *i.e.* a graph that encodes the library of components. Each type is represented as a *state*, and each component is represented as a directed *transition* from the component's input type to its output type. The synthesis problem then reduces to finding a *path* in the network that begins at the query's input type and ends at the output type [64]. To model components (functions) that take *multiple* inputs, we need only generalize the network to a *Petri net* which has *hyper*-transitions that link multiple input states with a single output. With this generalization, the synthesis problem can, once again, be solved by finding a path from the query's input types to the desired output yielding a scalable synthesis method for Java [27].

Challenge: Polymorphic Data and Components. Graph-based approaches crucially rely on the assumption that the size of the TTN is *finite* (and manageable). This assumption breaks down in the presence of *polymorphic* components that are ubiquitous in libraries for modern functional languages. (a) With polymorphic *datatypes* the set of types that might appear in a program is unbounded: for example, two type constructors `[]` and `Int` give rise to an *infinite* set of types (`Int`, `[Int]`, `[[Int]]`, *etc*). (b) Even if we bound the set of types, polymorphic *components* lead to a combinatorial explosion in the number of transitions: for example, the pair constructor with the type `a → b → (a,b)` creates a transition from *every pair of types* in the system. In other words, polymorphic data and components explode the size of the graph that must be searched, rendering synthesis intractable.

Type-Guided Abstraction Refinement. In this work we introduce *type-guided abstraction refinement* (TYGAR), a new approach to scalable type-directed synthesis over polymorphic datatypes and components. A high-level view of TYGAR is depicted in Fig. 1.1. The algorithm maintains an *abstract transition network* (ATN) that finitely *overapproximates* the infinite network comprising all monomorphic instances of the polymorphic data and components. We use existing SMT-based techniques to find a suitable path in the compact ATN, which corresponds to a candidate term. If the term is well-typed, it is returned as the solution. Due to the overapproximation, however, the ATN can contain *spurious* paths, which correspond to ill-typed

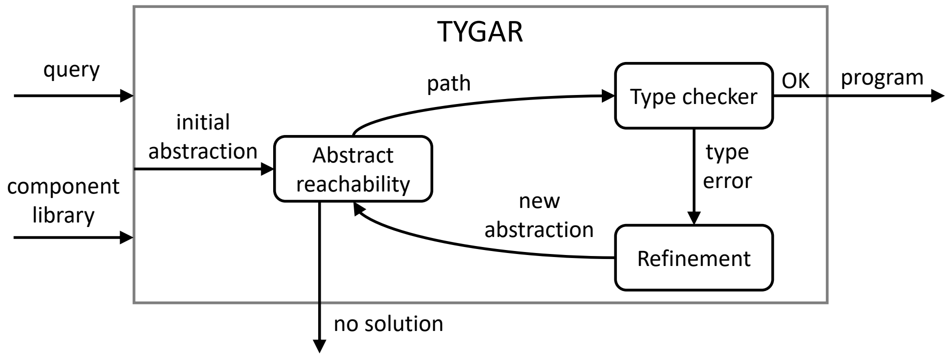


Figure 1.1. Overview of the TYGAR synthesis algorithm.

terms. In this case, the ATN is *refined* in order to exclude this spurious path, along with similar ones. We then repeat the search with the refined ATN until a well-typed solution is found. As such, TYGAR extends *synthesis using abstraction refinement* (SYNGAR) [103], from the domain of values to the domain of types. TYGAR's support for polymorphism also allows us to handle *higher-order* components, which take functions as input, by representing functions (arrows) as a binary type constructor. Similarly, TYGAR can handle Haskell's ubiquitous *type classes*, by following the dictionary-passing translation [100], which again, relies crucially on support for parametric polymorphism.

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

1. Abstract Typing. Our first contribution is a novel notion of *abstract typing* grounded in the framework of abstract interpretation [19]. Our abstract domain is parameterized by a finite collection of polymorphic types, each of which abstracts a potentially infinite set of ground instances. Given an abstract domain, we automatically derive an over-approximate type system, which we use to build the ATN. This is inspired by predicate abstraction [38], where the abstract domain is parameterized by a set of predicates, and abstract program semantics at different levels of detail can be derived automatically from the domain.

2. Type Refinement. Our second contribution is a new algorithm that, given a spurious program, refines the abstract domain so that the program no longer type-checks abstractly. To this end, the algorithm constructs a compact *proof of untypeability* of the program: it annotates each subterm with a type that is just precise enough to refute the program.

***3.* H+.** Our third contribution is an implementation of TYGAR in H+, a tool that takes as input a set of Haskell libraries and a type, and returns a ranked list of straight-line programs that have the desired type and can use any function from the provided libraries. To keep in line with HOOGLE's user interaction model familiar to Haskell programmers, H+ does not require any user input beyond the query type; this is in contrast to prior work on component-based synthesis [27, 89], where the programmer provides input-output examples to disambiguate their intent. This setting poses an interesting challenge: given that there might be hundreds of programs of a given type (including nonsensical ones like `head []`), how do we select just the *relevant* programs, likely to be useful to the programmer? We propose a novel mechanism for filtering out irrelevant programs using GHC's *demand analysis* [88] to eliminate terms where some of the inputs are unused.

We have evaluated H+ on a set of 44 queries collected from different sources (including HOOGLE and STACKOVERFLOW), using a set of popular Haskell libraries with a total of 291 components. Our evaluation shows that H+ is able to find a well-typed program for 43 out of 44 queries within the timeout of 60 seconds. It finds the first well-typed program within 1.4 seconds on average. In 32 out of 44 queries, the top five results contains a useful solution[2]. Further, our evaluation demonstrates that both abstraction and refinement are important for efficient synthesis. A naive approach that *does not use abstraction* and instead instantiates all polymorphic datatypes up to even a small depth of 1 yields a massive transition network, and is unable to solve any benchmarks within the timeout. On the other hand, an approach that uses a fixed small ATN but *no refinement* works well on simple queries, but fails to scale as the solutions get larger. Instead, the best performing search algorithm uses TYGAR to start with a small initial ATN and gradually extend it, up to a given size bound, with instances that are relevant for a given synthesis query.

[2]Unfortunately, ground truth solutions are not available for HOOGLE benchmarks; we judge usefulness by manual inspection.

```
-- | Value stored in the option
-- or default if the option is empty
fromMaybe :: α → Maybe α → α
-- | All values from a list of options
catMaybes :: List (Maybe α) → List α
-- | Head of the list
-- or empty option if the list is empty
listToMaybe :: List α → Maybe α
```

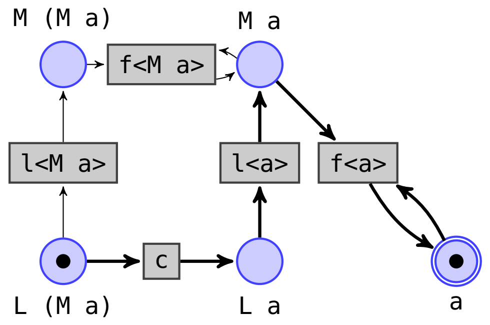


Figure 1.2. (left) A tiny component library. (right) A Type Transition Net for this library and query `a → List (Maybe a) → a`. The transitions `l<a>`, `f<a>` (resp. `l<M a>`, `f<M a>`) correspond to the polymorphic instances of the components `listToMaybe`, `fromMaybe` at type `a` (resp. `M a`).

1.2 Background and Overview

We start with some examples that illustrate the prior work on component-based synthesis that H+ builds on (Sec. 1.2.1), the challenges posed by polymorphic components, and our novel techniques for addressing those challenges.

1.2.1 Synthesis via Type Transition Nets

The starting point of our work is SyPet [27], a component-based synthesizer for Java. Let us see how SyPet works by using the example query from the introduction: `a → [Maybe a] → a`. For the sake of exposition, we assume that our library only contains three components listed in Fig. 1.2 (left). Hereafter, we will use Greek letters $\alpha, \beta, \ldots$ to denote *existential type variables*—*i.e.* the type variables of components, which have to be instantiated by the synthesizer—as opposed to $a, b, \ldots$ for *universal type variables* found in the *query*, which, as far as the synthesizer is concerned, are just nullary type constructors. Since SyPet does not support polymorphic components, let us assume for now that an oracle provided us with a small set of monomorphic types that suffice to answer this query, namely, `a`, `Maybe a`, `Maybe (Maybe a)`, `[a]`, and `[Maybe a]`. For the rest of this section, we abbreviate the names of components and type constructors to their first letter (for example, we will write `L (M a)` for `[Maybe a]`) and refer to the query arguments as `x1`, `x2`.

Components as Petri Nets. SyPet uses a Petri-net representation of the search space, which we refer to as the *type transition net* (TTN). The TTN for our running example is shown in Fig. 1.2 (right). Here *places* (circles) correspond to types, *transitions* (rectangles) correspond to components, and *edges* connect components with their input and output types. Since a component might require multiple inputs of the same type, edges can be annotated with *multiplicities* (the default multiplicity is 1). A *marking* of a TTN assigns a non-negative number of *tokens* to every place. The TTN can step from one marking to the next by *firing* a transition: if the input places of a transition have sufficiently many tokens, the transition can fire, consuming those input tokens and producing a token in the output place. For example, given the marking in Fig. 1.2, transition `c` can fire, consuming the token in `L (M a)` and producing one in `L a`; however transition `f<a>` cannot fire as there is no token in `M a`.

Synthesis via Petri-Net Reachability. Given a synthesis query $T_1 \to \ldots \to T_n \to T$, we set the *initial marking* of the TTN to contain one token for each input type T_i, and the *final marking* to contain a single token in the type T. The synthesis problem then reduces to finding a *valid path*, *i.e.* a sequence of fired transitions that gets the net from the initial marking to the final marking. Fig. 1.2 shows the initial marking for our query, and also indicates the final marking with a double border around the return type `a` (recall that the final marking of a TTN always contains a single token in a given place). The final marking is reachable via the path $[\mathtt{c}, \mathtt{l}, \mathtt{f}]$, marked with thick arrows, which corresponds to a well-typed program `f x1 (l (c x2))`. In general, a path might correspond to multiple programs—if several tokens end up in the same place at any point along the path—of which at least one is guaranteed to be well-typed; the synthesizer can then find the well-typed program using explicit or symbolic enumeration.

1.2.2 Polymorphic Synthesis via Abstract Type Transition Nets

Libraries for modern languages like Haskell provide highly polymorphic components that can be used at various different instances. For example, our universe contains three type constructors—`a`, `L`, and `M`—which can give rise to infinitely many types, so creating a place for each type is out of question. Even if we limit ourselves to those constructors that are reachable from the query types by following the components, we might still end up with an infinite set of types: for example, following `head :: List` $\alpha \to \alpha$ backwards from `a` yields `L a`, `L (L a)`, and so on. This poses a challenge for Petri-net based synthesis: *which finite set of (monomorphic) instances do we include in the TTN?*

On the one hand, we have to be careful not to include *too many* instances. In the presence of polymorphic components, these instances can explode the number of transitions. Fig. 1.2 illustrates this for the `f` and `l` components, each giving rise to two transitions, by instantiating their type variable α with two different TTN places, `a` and `Maybe a`. This proliferation of transitions is especially severe for components with multiple type variables. On the other hand, we have to be careful not to include *too few* instances. We cannot, for example, just limit ourselves to the monomorphic types that are explicitly present in the query (`a` and `L (M a)`), as this will preclude the synthesis of terms that generate intermediate values of some other type, *e.g.* `L a` as returned by the component `c`, thereby preventing the synthesizer from finding solutions.

Abstract Types. To solve this problem, we introduce the notion of an *abstract type*[3], which stands for (infinitely) many monomorphic instances. We represent abstract types simply as polymorphic types, *i.e.* types with free type variables. For example, the abstract type τ stands for the set of all types, while `L` τ stands for the set $\{\texttt{L}\ t \mid t \in \mathit{Type}\}$. This representation supports different levels of detail: for example, the type `L (M a)` can be abstracted into itself, `L (M` τ`)`, `L` τ, or τ.

Abstract Transition Nets. A Petri net constructed out of abstract types, which we dub an *abstract transition net* (ATN), can finitely represent all types in our universe, and hence all possible solutions to the synthesis problem. The ATN construction is grounded in the theory of abstract interpretation and ensures that the net *soundly over-approximates* the concrete type system, *i.e.* that every well-typed program corresponds to some valid path through the ATN. Fig. 1.3 (2) shows the ATN for our running example with places τ, `L` τ and `a`. In this ATN, the *rightmost*

[3]Not to be confused with existing notions of *abstract data type* and *abstract class*. We use "abstract" here is the sense of abstract interpretation [19], *i.e.* an abstraction of a set of concrete types.

`f` transition takes `a` and τ as inputs and returns `a` as output. This transition represents the set of monomophic types $\{\texttt{a} \to t \to \texttt{a} \mid t \in \mathit{Type}\}$ and *over-approximates* the set of instances of `f` where the first argument unifies with `a` and the second argument unifies with τ (which in this case is a singleton set $\{\texttt{a} \to \texttt{M a} \to \texttt{a}\}$). Due to the over-approximation, some of the ATN's paths yield *spurious* ill-typed solutions. For example, via the highlighted path, this ATN produces the term `f x1 (l x2)`, which is ill-typed since the arguments to `f` have the types `a` and `M (M a)`.

How do we pick the right level of detail for the ATN? If the places are too abstract, there are too many spurious solutions, leading, in the limit, to a brute-force enumeration of programs. If the places are too concrete, the net becomes too large, and the search for valid paths is too slow. Ideally, we would like to pick a minimal set of abstract types that only make distinctions pertinent to the query at hand.

Type-Guided Abstraction Refinement. H+ solves this problem using an iterative process we call *type-guided abstraction refinement* (TYGAR) where an initial coarse abstraction is incrementally refined using the information from the type errors found in spurious solutions. Next, we illustrate TYGAR using the running example from Fig. 1.3.

Iteration 1. We start with the coarsest possible abstraction, where all types are abstracted to τ, yielding the ATN in Fig. 1.3 (1). The shortest valid path is just [`f`], which corresponds to two programs: `f x1 x2` and `f x2 x1`. Next, we type-check these programs to determine whether they are valid or spurious. During type checking, we compute the principal type of each sub-term and propagate this information bottom-up through the AST; the resulting *concrete typing* is shown in red at the bottom of Fig. 1.3 (1). Since both candidate programs are ill-typed (as indicated by the annotation $\perp$ at the root of either AST), the current path is spurious. Although we could simply enumerate more valid paths until we find a well-typed program, such brute-force enumeration does not scale with the number of components. Instead, we refine the abstraction so that this path (and hopefully many similar ones) becomes invalid.

Our refinement uses the type error information obtained while type-checking the spurious

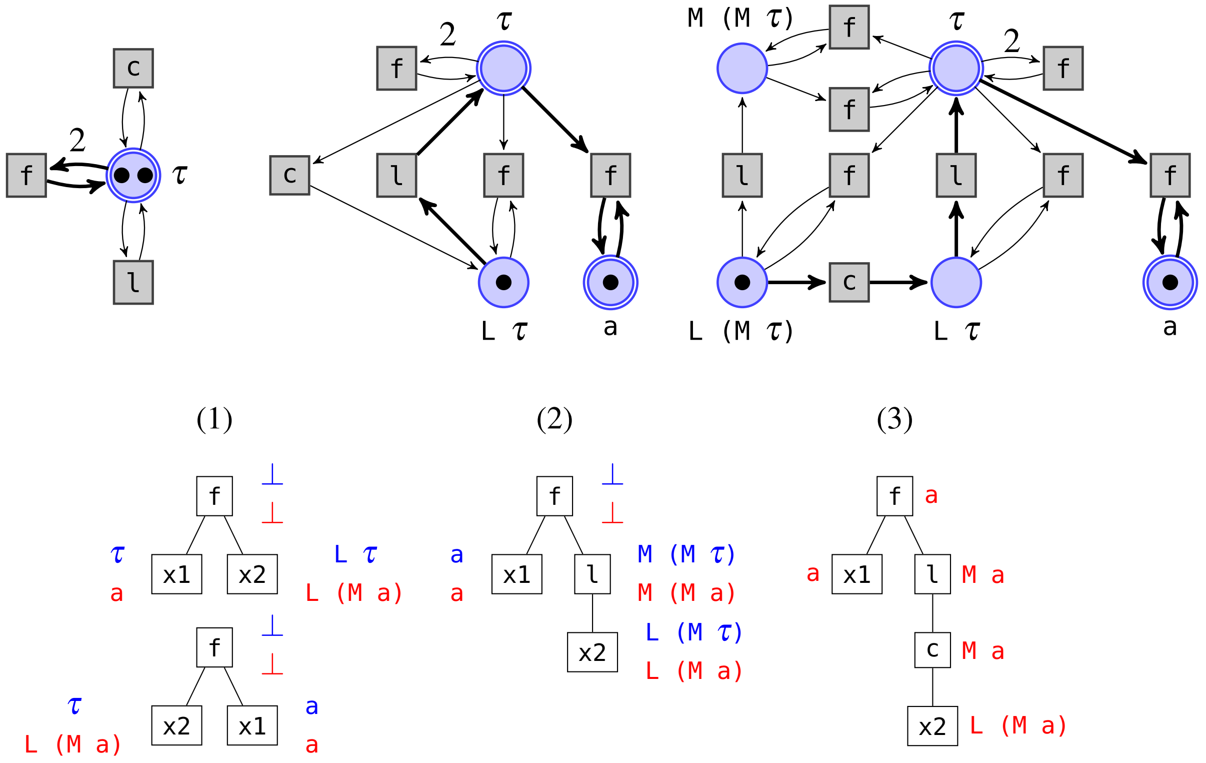


Figure 1.3. Three iterations of abstraction refinement: ATNs (above) and corresponding solutions (below). Some irrelevant transitions are omitted from the ATNs for clarity. Solutions 1 and 2 are spurious, solution 3 is valid. Each solution is annotated with its concrete typing (in red); each spurious solution is additionally annotated with its proof of untypeability (in blue). These blue types are added to the ATN in the next iteration.

programs. Consider `f x1 x2`: the program is ill-typed because the concrete type of `x2`, `L (M a)`, does not unify with the second argument of `f`, `M` α. To avoid making this type error in the future, we need to make sure that the *abstraction* of `L (M a)` also fails to unify with `M` α. To this end, we need to extend our ATN with new abstract types, that suffice to reject the program `f x1 x2`. These new types will update the ATN with new places that will *reroute* the transitions so that the path that led to the term `f x1 x2` is no longer feasible. We call this set of abstract types a *proof of untypeability* of the program. We could use `x2`'s concrete type `L (M a)` as the proof, but we want the proof to be as general as possible, so that it can reject more programs. To compute a better proof, the TYGAR algorithm *generalizes* the concrete typing of the spurious program, repeatedly weakening concrete types with fresh variables while still preserving untypeability. In our example, the generalization step yields τ and `L` τ (see blue annotations in Fig. 1.3). This general proof also rejects other programs that use a list as the second argument to `f`, such as `f x1 (c x2)`. Adding the types from the untypeability proofs of both spurious programs to the ATN results in a refined net shown in Fig. 1.3 (2).

Iteration 2. The new ATN in Fig. 1.3 (2) has no valid paths of length one, but has the (highlighted) path [`l`, `f`] of length two, which corresponds to a single program `f x1 (l x2)` (since the two tokens never cross paths). This program is ill-typed, so we refine the abstraction based on its untypeability, as depicted at the bottom of Fig. 1.3 (2). To compute the proof of untypeability, we start by generalizing the concrete types of `f`'s arguments as much as possible as long as the application remains ill-typed, arriving at the types `a` and `M (M` τ`)`. Generalization then propagates top-down through the AST: in the next step, we compute the most general abstraction for the type of `x2` such that `l x2` has type `M (M` τ`)`. The generalization process stops at the leaves of the AST (or alternatively when the type of some node cannot be generalized). Adding the types `M (M` τ`)` and `L (M` τ`)` from the untypeability proof to the ATN leads to the net in Fig. 1.3 (3).

Iteration 3. The shortest valid path in the third ATN is [`c`, `l`, `f`], corresponding to a well-typed program `f x1 (l (c x2))` (bottom of Fig. 1.3 (3)), which we return as the solution.

1.2.3 Pruning Irrelevant Solutions via Demand Analysis

Using a query type as the sole input to synthesis has its pros and cons. On the one hand, types are programmer-friendly: unlike input-output examples, which often become verbose and cumbersome for data other than lists, types are concise and versatile, and their popularity with Haskell programmers is time-tested by the HOOGLE API search engine. On the other hand, a query type only partially captures the programmer's intent; in other words, not all well-typed programs are equally desirable. In our running example, the program `\x1 x2` $\rightarrow$ `x1` has the right type, but it is clearly uninteresting. Hence, the important challenge for H+ is: how do we filter out uninteresting solutions *without* requiring additional input from the user?

Relevant Typing. SYPET offers an interesting approach to this problem: they observe that a programmer is unlikely to include an argument in a query if this argument is not required for the solution. To leverage this observation, they propose to use a *relevant* type system [78], which requires each variable to be used *at least once*, making programs like `\x1 x2` $\rightarrow$ `x1` ill-typed. TTNs naturally enforce relevancy during search: in fact, TTN reachability as described so far encodes a stricter *linear* type system, where all arguments must be used *exactly once*. This requirement can be relaxed by adding special "copy" transitions that consume one token from a place and produce two token in the same place.

Demand Analysis. Unfortunately, with expressive polymorphic components the synthesizer discovers ingenious ways to circumvent the relevancy requirement. For example, the terms `fst (x1, x2)`, `const x1 x2`, and `fromLeft x1 (Right x2)` are all functionally equivalent to `x1`, even though they satisfy the letter of relevant typing. To filter out these solutions, we use GHC's *demand analysis* [88] to post-process solutions returned by the ATN and filter out those with unused variables. Demand analysis is a whole-program analysis that peeks inside the component implementation, and hence is able to infer in all three cases above that the variable `x2` is unused. As we show in Sec. 1.6, demand analysis significantly improves the quality of solutions.

```
-- | Function application
($) :: (α → β) → α → β
-- | List with n copies of a value
replicate :: Int → α → [α]
-- | Fold a list
foldr :: (α → β → β) → β → [α] → β
-- | Value stored in the option
fromJust :: Maybe α → α
-- | Lookup element by key
lookup :: Eq α => α → [(α, β)] → Maybe β
```

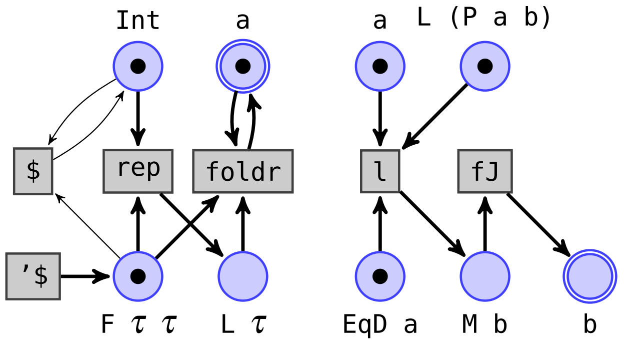


Figure 1.4. (left) A library with higher-order functions and type-class constraints. (center) Fragment of an ATN for the query `(a → a) → a → Int → a`. (right) Fragment of an ATN for the query `Eq a => [(a,b)] → a → b`.

1.2.4 Higher-Order Functions

Next we illustrate how ATNs scale up to account for higher-order functions and type classes, using the component library in Fig. 1.4 (left), which uses both of these features.

Example: Iteration. Suppose the user poses a query `(a → a) → a → Int → a`, with the intention to apply a function *g* to an initial value *x* some number of times *n*. Perhaps surprisingly, this query can be solved using components in Fig. 1.4 by creating a list with *n* copies of *g*, and then *folding* function application over that list with the seed *x* – that is, via the term `\g x n → foldr ($) x (replicate n g)`.

Can we generate this solution using an ATN? As described so far, ATNs only assign places to base (non-arrow) types, and hence cannot synthesize terms that use higher-order components, such as the application of `foldr` to the function `($)` above. Initially, we feared that supporting higher-order components would require generating *lambda terms* within the Petri net (to serve as their arguments) which would be beyond the scope of this work. However, in common cases like our example, the higher-order argument can be written as a single variable (or component). Hence, the full power of lambda terms is not required.

HOF Arguments via Nullary Components. We support the common use case — where higher-order arguments are just components or applications of components — simply by desugaring a higher-order library into a first-order library supported by ATN-based synthesis. To this end, we (1) introduce a binary type constructor `F` α β to represent arrow types as if they were base types; and (2) for each component `c :: B1` $\rightarrow$ `...` $\rightarrow$ `Bn` $\rightarrow$ `B` in the original library, we add a *nullary* component `'c :: F B1 (... F Bn B)`. Intuitively, an ATN distinguishes between functions it *calls* (represented as transitions) and functions it uses as *arguments* to other functions (represented as tokens in corresponding `F` places).

Fig. 1.4 (center) depicts a fragment of an ATN for our example. Note that the `($)` component gives rise both to a binary transition `$`, which we would take if we were to apply this component, and a nullary transition `'$`, which is taken by our solution, since `($)` is used as an argument to `foldr`. Since `F` is just an ordinary type constructor as far as the ATN is concerned, all existing abstraction and refinement mechanisms apply to it unchanged: for example, in Fig. 1.4 both `a` $\rightarrow$ `a` and `(a` $\rightarrow$ `a)` $\rightarrow$ `a` $\rightarrow$ `a` are abstracted into the same place `F` τ τ.

Completeness via Point-Free Style. While our method was inspired by the common use case where the higher-order arguments were themselves components, note that with a sufficiently rich component library, *e.g.* one that has representations of the `S`, `K` and `I` combinators, our method is *complete* as every term that requires an explicit lambda-subterm for a function argument can now be written in a point-free style, only using variables, components and their applications.

1.2.5 Type Classes

Type classes are widely used in Haskell to support *ad-hoc* polymorphism [100]. For example, consider the type of component `lookup` in Fig. 1.4: this function takes as input a key *k* of type α and a list of key-value pairs of type `[(`α`, `β`)]`, and returns the value that corresponds to *k*, if one exists. In order to look up *k*, the function has to compare keys for equality; to this end, its signature imposes a *bound* `Eq` α on the type of keys, enforcing that any concrete key type be an instance of the *type class* `Eq` and therefore be equipped with a definition of equality.

Type classes are implemented by a translation to parametric polymorphism called *dictionary passing*, where each class is translated into a record whose fields implement the different functions supported by the type class. Happily, H+ can use dictionary passing to desugar synthesis with type classes into a synthesis problem supported by ATNs. For example, the type of `lookup` is desugared into an unbounded type with an extra argument: `EqD` $\alpha \rightarrow \alpha \rightarrow$ `[(`α`,`β`)]` $\rightarrow \beta$. Here `EqD` α, is a *dictionary*: a record datatype that stores the implementation of equality on α; the exact definition of this datatype is unimportant, we only care whether `EqD` α for a given α is inhabited.

Example: Key-Value Lookup. As a concrete example, suppose the user wants to perform the lookup operation in a key-value list *assuming* the key is present. This task can be expressed as the type query `Eq a => [(a,b)]` $\rightarrow$ `a` $\rightarrow$ `b`. The intended solution to this query is `\xs k` $\rightarrow$ `fromJust (lookup k xs)`, *i.e.* look up the key and then extract the value from the option, assuming it is nonempty. A fragment of an ATN for this query is shown in Fig. 1.4 (right). Note that the transition `l`—the instance of `lookup` with $\alpha \mapsto$ `a`, $\beta \mapsto$ `b`—has `EqD a` as one of its incoming edges. This corresponds to our intuition about type classes: in order to fire `l`, the ATN first has to prove that `a` satisfies `Eq`, or in other words, that `EqD a` is inhabited. In this case, the proof is trivial: because the query type is also desugared in the same way, the initial marking contains a token in `EqD a`[4]. A welcome side-effect of relevant typing is that any solution *must use* the token in `EqD a`, which matches our intuition that the user would not specify the bound `Eq a` if they did not mean to compare keys for equality. This example illustrates that the combination of (bounded) polymorphism and relevant typing gives users a surprisingly powerful mechanism to disambiguate their intent. Given the query above (and a library of 291 components), H+ returns the intended solution as the first result. In contrast, given a monomorphic variant of this query `[(Int, b)]` $\rightarrow$ `Int` $\rightarrow$ `b` (where the key type is just an `Int`) H+ produces a flurry of irrelevant results, such as `\xs k` $\rightarrow$ `snd (xs !! k)`, which uses `k` as an *index* into the list, and not as a key as we intended.

[4]As we explain in Sec. 1.5.1, dictionaries can also be inhabited via instances and functional dependencies.

1.3 Abstract Type Checking

Next, we formally define the syntax of our target language λ_H and its type system, and use the framework of *abstract interpretation* to develop an algorithmic *abstract* type system for λ_H. This framework allows us to parameterize the checker by the desired level of detail, crucially enabling our novel TYGAR synthesis algorithm formalized in Sec. 1.4.

1.3.1 The λ_H Language

λ_H is a simple first-order language with a prenex-polymorphic type system, whose syntax and typing rules are shown in Fig. 1.5. We stratify the terms into *application* terms which comprise variables x, library components c and applications; and *normal-form* terms which are lambda-abstractions over application terms.

The *base* types B include type variables τ, as well as applications of a type constructor to zero or more base types $C\,\overline{B}$. We write $\overline{X}$ to denote zero or more occurrences of a syntactic element X. Types T include base types and first-order function types (with base-typed arguments). Syntactic categories b and t are the ground counterparts to B and T (*i.e.* they contain no type variables). A *component library* Λ is a finite map from a set of components c to the components' poly-types. A *typing environment* Γ is a map from variables x to their ground base types. A *substitution* $\sigma = [\tau_1 \mapsto B_1, \dots, \tau_n \mapsto B_n]$ is a mapping from type variables to base types that maps each τ_i to B_i and is identity elsewhere. We write σT to denote the application of σ to type T, which is defined in a standard way.

A *typing judgment* $\Lambda; \Gamma \vdash E :: t$ is only defined for ground types t. Polymorphic components are instantiated into ground monotypes by the COMP rule, which angelically picks ground base types to substitute for all the universally-quantified type variables in the component signature (the rule implicitly requires that σT be ground).

Syntax

e	$::= x \mid c \mid e\ e$	*Application Terms*
E	$::= e \mid \lambda x.E$	*Normal-Form Terms*
b	$::= C\ \overline{b}$	*Ground Base Types*
t	$::= b \mid b \rightarrow t$	*Ground Types*
B	$::= \tau \mid C\ \overline{B}$	*Base Types*
T	$::= B \mid B \rightarrow T$	*Types*
P	$::= \overline{\forall \tau}.T$	*Polytypes*
Γ	$::= \cdot \mid x : b, \Gamma$	*Environments*
σ	$::= [\overline{\tau \mapsto B}]$	*Substitutions*

Typing $\boxed{\Lambda; \Gamma \vdash E :: t}$

$$\text{T-VAR}\ \frac{\Gamma(x) = b}{\Lambda; \Gamma \vdash x :: b}$$

$$\text{T-COMP}\ \frac{\Lambda(c) = \overline{\forall \tau}.T}{\Lambda; \Gamma \vdash c :: \sigma T}$$

$$\text{T-APP}\ \frac{\Lambda; \Gamma \vdash e_1 :: b \rightarrow t \qquad \Lambda; \Gamma \vdash e_2 :: b}{\Lambda; \Gamma \vdash e_1\ e_2 :: t}$$

$$\text{T-FUN}\ \frac{\Lambda; \Gamma, x : b \vdash E :: t}{\Lambda; \Gamma \vdash \lambda x.E :: b \rightarrow t}$$

Figure 1.5. λ_H: syntax and declarative type system.

1.3.2 Type Checking as Abstract Interpretation

Type subsumption lattice. We say that type T' *is more specific than* type T (or alternatively, that T *is more general than* or *subsumes* T') written $T' \sqsubseteq T$, iff there exists σ such that $T' = \sigma T$. The relation $\sqsubseteq$ is a partial order on types. For example, in a library with two nullary type constructors `A` and `B`, and a binary type constructor `P`, we have $\texttt{P A B} \sqsubseteq \texttt{P}\ \alpha\ \texttt{B} \sqsubseteq \texttt{P}\ \alpha\ \beta \sqsubseteq \tau$. This partial order induces an equivalence relation $T_1 \equiv T_2 \triangleq T_1 \sqsubseteq T_2 \wedge T_2 \sqsubseteq T_1$ (equivalence up to variable renaming). The order (and equivalence) relation extends to substitutions in a standard way: $\sigma' \sqsubseteq \sigma \triangleq \exists \rho. \forall \tau. \sigma' \tau = \rho \sigma \tau$.

We augment the set of types with a special bottom type $\perp$ that is strictly more specific than every other base type; we also consider a bottom substitution $\sigma_\perp$ and define $\sigma_\perp B = \perp$ for any B. A *unifier* of B_1 and B_2 is a substitution σ such that $\sigma B_1 = \sigma B_2$; note that $\sigma_\perp$ is a unifier for any two types. The most general unifier (MGU) is unique up to $\equiv$, and so, by slight abuse of notation, we write it as a function $\mathsf{mgu}(B_1, B_2)$. We write $\mathsf{mgu}(\overline{B_1, B_2})$ for the MGU of a sequence of type pairs, where the MGU of an empty sequence is the identity substitution ($\mathsf{mgu}(\cdot) = []$). The *meet* of two base types is defined as $B_1 \sqcap B_2 = \sigma B_1 (= \sigma B_2)$, where $\sigma = \mathsf{mgu}(B_1, B_2)$. For example, $\texttt{P}\ \alpha\ \texttt{B} \sqcap \texttt{P A}\ \beta = \texttt{P A B}$ while $\texttt{P}\ \alpha\ \texttt{B} \sqcap \texttt{P}\ \beta\ \texttt{A} = \perp$. The *join* of two base types can be defined as their anti-unifier, but we elide a detailed discussion as joins are not required for our purposes.

We write $\mathbf{B}_\perp = \mathbf{B} \cup \{\perp\}$ for the set of base types augmented with $\perp$. Note that

$\langle \mathbf{B}_\perp, \sqsubseteq, \sqcup, \sqcap \rangle$ is a lattice with bottom element $\perp$ and top element τ and is isomorphic to [80]'s *subsumption lattice* on first-order logic terms.

Type Transformers. A component signature can be interpreted as a partial function that maps (tuples of) ground types to ground types. For example, a component `l :: ` $\forall\, \beta.$`L` $\beta \to$ `M` β maps `L A` to `M A`, `L (M A)` to `M (M A)`, and `A` to $\perp$. This gives rise to *type transformer* semantics for components, which is similar to predicate transformer semantics in predicate abstraction and SYNGAR [103], but instead of being designed by a domain expert can be derived automatically from the component signatures.

More formally, we define a *fresh instance* of a polytype $\mathsf{fresh}(\forall\overline{\tau}.T) \triangleq [\overline{\tau \mapsto \tau'}]T$, where $\overline{\tau'}$ are fresh type variables. Let c be a component and $\mathsf{fresh}(\Lambda(c)) = \overline{B'_i} \to B'$; then a *type transformer* for c is a function $[\![c]\!]_\Lambda \colon \overline{\mathbf{B}_\perp} \to \mathbf{B}_\perp$ defined as follows:

$$[\![c]\!]_\Lambda(\overline{B_i}) = \sigma B' \quad \text{where } \sigma = \mathsf{mgu}(\overline{B_i, B'_i})$$

We omit the subscript Λ where the library is clear from the context. For example, for the component `l` above: $[\![\mathtt{l}]\!](\mathtt{L\ (M\ }\tau\mathtt{)}) = \mathtt{M\ (M\ }\tau\mathtt{)}$, $[\![\mathtt{l}]\!](\tau) = \mathtt{M}\ \tau_1$ (where τ_1 is a fresh type variable), and $[\![\mathtt{l}]\!](\mathtt{A}) = \perp$ (because $\mathsf{mgu}(\mathtt{L}\ \tau_2, \mathtt{A}) = \sigma_\perp$). We can show that this type transformer is *monotone*: applying it to more specific types yield a more specific type. The transformer is also *sound* in the sense that in any concrete type derivation where the argument to `l` is more specific than some B, its result is guaranteed to be more specific than $[\![\mathtt{l}]\!](B)$.

Lemma 1.3.1 (Monotonicity of Substitution)**.** *If $\sigma' \sqsubseteq \sigma$ then $\sigma' B \sqsubseteq \sigma B$.*

Lemma 1.3.2 (Monotonicity of Unification)**.** *If $B_1 \sqsubseteq B_2$ then* $\mathsf{mgu}(B, B_1) \sqsubseteq \mathsf{mgu}(B, B_2)$

Lemma 1.3.3 (Monotonicity of Type Transformers)**.** *For any component c and any types $\overline{B^1_i}$ and $\overline{B^2_i}$, such that $\overline{B^1_i \sqsubseteq B^2_i}$, we have $[\![c]\!](\overline{B^1_i}) \sqsubseteq [\![c]\!](\overline{B^2_i})$.*

Proof. Let $\mathsf{fresh}(\Lambda(c)) = \overline{B_i} \to B$. By the definition of type transformers, we have

$$\begin{aligned} [\![c]\!](\overline{B_i^1}) &= \sigma_1 B & \qquad \sigma_1 &= \mathsf{mgu}(\overline{B_i, B_i^1}) \\ [\![c]\!](\overline{B_i^2}) &= \sigma_2 B & \qquad \sigma_2 &= \mathsf{mgu}(\overline{B_i, B_i^2}) \end{aligned}$$

Since $\overline{B_i^1 \sqsubseteq B_i^2}$, we have $\sigma_1 \sqsubseteq \sigma_2$ by Lemma 1.3.2, and hence $\sigma_1 B \sqsubseteq \sigma_2 B$ by Lemma 1.3.1. □

Lemma 1.3.4 (Soundness of Type Transformers)**.** *If* $\mathsf{fresh}(\Lambda(c)) = \overline{B_i} \to B$ *and* $\overline{\sigma B_i \sqsubseteq B_i'}$ *then* $\sigma B \sqsubseteq [\![c]\!](\overline{B_i'})$.

Proof. By Lemma 1.3.3, since $\overline{\sigma B_i \sqsubseteq B_i'}$, we have $[\![c]\!](\overline{\sigma B_i}) \sqsubseteq [\![c]\!](\overline{B_i'})$. But $[\![c]\!](\overline{\sigma B_i}) = \sigma B$, because $\mathsf{mgu}(B, \sigma B) \equiv \sigma$. Hence $\sigma B \sqsubseteq [\![c]\!](\overline{B_i'})$ as desired. □

Bidirectional Typing. We can use type transformers to define algorithmic type checking for λ_H, as shown in Fig. 1.6. For now, ignore the parts of the rules highlighted in red, or, in other words, assume that $\alpha_{\mathcal{A}}$ is the identity function; the true meaning of this function is explained in the next section. As is standard in bidirectional type checking [79], the type system is defined using two judgments: the *inference judgment* $\Lambda; \Gamma \vdash e \Longrightarrow B$ generates the (base) type B from the term e, while the *checking judgment* $\Lambda; \Gamma \vdash E \Longleftarrow t$ checks E against a known (ground) type t. Algorithmic typing assumes that the term is in η-long form, *i.e.* there are no partial applications. During type checking, the outer λ-abstractions are handled by the checking rule C-FUN, and then the type of inner application term is inferred and compared with the given type b in C-BASE.

The only interesting case is the inference rule I-APP, which handles (uncurried) component applications using their corresponding type transformers. Nullary components are handled by the same rule (note that in this case $[\![c]\!] = \mathsf{fresh}(\Lambda(c))$). This type system is algorithmic, because we have eliminated the angelic choice of polymorphic component instantiations (recall the T-COMP rule in the declarative type system). Moreover, type inference for application terms can be thought of as abstract interpretation, where the abstract domain is the type subsumption

(Abstract) Type Inference $\boxed{\Lambda;\Gamma \vdash_{\mathcal{A}} e \Longrightarrow B}$

$$\text{I-VAR}\;\frac{\Gamma(x) = b}{\Lambda;\Gamma \vdash_{\mathcal{A}} x \Longrightarrow \alpha_{\mathcal{A}}(b)} \qquad \text{I-APP}\;\frac{\overline{\Lambda;\Gamma \vdash_{\mathcal{A}} e_i \Longrightarrow B_i}}{\Lambda;\Gamma \vdash_{\mathcal{A}} c\ \overline{e_i} \Longrightarrow \alpha_{\mathcal{A}}\left(\llbracket c \rrbracket(\overline{B_i})\right)}$$

(Abstract) Type Checking $\boxed{\Lambda;\Gamma \vdash_{\mathcal{A}} E \Longleftarrow t}$

$$\text{C-FUN}\;\frac{\Lambda;\Gamma, x : b \vdash_{\mathcal{A}} E \Longleftarrow t}{\Lambda;\Gamma \vdash_{\mathcal{A}} \lambda x.E \Longleftarrow b \to t} \qquad \text{C-BASE}\;\frac{\Lambda;\Gamma \vdash_{\mathcal{A}} e \Longrightarrow B \quad b \sqsubseteq B}{\Lambda;\Gamma \vdash_{\mathcal{A}} e \Longleftarrow b}$$

Figure 1.6. Abstract type checking for λ_H. Treating $\alpha_{\mathcal{A}}$ as the identity function yields concrete type checking.

lattice: for any application term e, the inference computes its "abstract value" B (known in type inference literature as its *principal type*). We can show that the algorithmic system is sound and complete with respect to the declarative one.

Theorem 1.3.5 (Type Checking is Sound and Complete)**.** $\Lambda;\cdot \vdash E :: t$ *iff* $\Lambda;\cdot \vdash E \Longleftarrow t$.

1.3.3 Abstract Typing

The algorithmic typing presented so far is just a simplified version of Hindley-Milner type inference. However, casting type inference as abstract interpretation gives us the flexibility to tune the *precision* of the type system by restricting the abstract domain to a *sub-lattice* of the full type subsumption lattice. This is similar to predicate abstraction, where precision is tuned by restricting the abstract domain to boolean combinations of a finite set of predicates.

Abstract Cover. An *abstract cover* $\mathcal{A} = \{A_1, \ldots, A_n\}$ is a set of base types $A_i \in \mathbf{B}_\perp$ that contains τ and $\perp$, and is a sub-lattice of the type subsumption lattice (importantly, it is closed under $\sqcap$). For example, in a library with a nullary constructor `A` and two unary constructors `L` and `M`, $\mathcal{A}_0 = \{\tau, \perp\}$, $\mathcal{A}_1 = \{\tau, \texttt{A}, \texttt{L}\ \tau, \perp\}$, and $\mathcal{A}_2 = \{\tau, \texttt{A}, \texttt{L}\ \tau, \texttt{L}\ (\texttt{M}\ \tau), \texttt{M}\ (\texttt{M}\ \tau), \perp\}$ are abstract covers. Note that in a cover, the scope of a type variable is each individual base type, so the different instances of τ above are unrelated. We say that an abstract cover $\mathcal{A}'$ *refines* a cover $\mathcal{A}$ ($\mathcal{A}' \preceq \mathcal{A}$) if $\mathcal{A}$ is a sub-lattice of $\mathcal{A}'$. In the example above, $\mathcal{A}_2 \preceq \mathcal{A}_1 \preceq \mathcal{A}_0$.

Abstraction function. Given an abstract cover $\mathcal{A}$, the *abstraction* $\alpha_{\mathcal{A}} \colon \mathbf{B}_\bot \to \mathbf{B}_\bot$ of a base type B is defined as the most specific type in $\mathcal{A}$ that subsumes B:

$$\alpha_{\mathcal{A}}(B) = A \in \mathcal{A} \text{ such that } B \sqsubseteq A \text{ and } \forall A' \in \mathcal{A}.B \sqsubseteq A' \Rightarrow A \sqsubseteq A'$$

We can show that $\alpha_{\mathcal{A}}(B)$ is unique, because $\mathcal{A}$ is closed under meet. In abstract interpretation, it is customary to define a dual *concretization* function. In our case, the abstract domain $\mathcal{A}$ is a sub-lattice of the concrete domain $\mathbf{B}_\bot$, and hence our concretization function is the identity function *id*. It is easy to show that $\alpha_{\mathcal{A}}$ and *id* form a *Galois insertion*, because $B \sqsubseteq id(\alpha_{\mathcal{A}}(B))$ and $A = \alpha_{\mathcal{A}}(id(A))$ both hold by definition of $\alpha_{\mathcal{A}}$.

Abstract Type Checking. Armed with the definition of abstraction function, let us now revisit Fig. 1.6 and consider the highlighted parts we omitted previously. The two abstract typing judgments—for checking and inference—are parameterized by the abstract cover. The only interesting changes are in the *abstract type inference* judgment $\Lambda;\Gamma \vdash_{\mathcal{A}} e \Longrightarrow B$, which applies the abstraction function to the inferred type at every step. For example, recall the covers $\mathcal{A}_1$ and $\mathcal{A}_2$ defined above, and consider a term `l xs` where $\Lambda(\texttt{l}) = \forall\beta.\texttt{L}\ \beta \to \texttt{M}\ \beta$ and $\Gamma(\texttt{xs}) = \texttt{L (M A)}$. Then in $\mathcal{A}_1$ we infer $\Lambda;\Gamma \vdash_{\mathcal{A}_1} \texttt{l xs} \Longrightarrow \tau$, since $\alpha_{\mathcal{A}_1}(\texttt{L (M A)}) = \texttt{L}\ \tau$ and $[\![\texttt{l}]\!](\texttt{L}\ \tau) = \texttt{M}\ \tau$, but $\texttt{M}\ \tau$ is abstracted to τ. However, in $\mathcal{A}_2$ we infer $\Lambda;\Gamma \vdash_{\mathcal{A}_2} \texttt{l xs} \Longrightarrow \texttt{M (M}\ \tau\texttt{)}$, since $\alpha_{\mathcal{A}_2}(\texttt{L (M A)}) = \texttt{L (M}\ \tau\texttt{)}$, and $[\![\texttt{l}]\!](\texttt{L (M}\ \tau\texttt{)}) = \texttt{M (M}\ \tau\texttt{)}$, which is abstracted to itself.

We can show that abstraction preserves typing: *i.e.* E has type t in an abstraction $\mathcal{A}$ whenever it has type t in a *more refined* abstraction $\mathcal{A}' \preceq \mathcal{A}$:

Theorem 1.3.6 (Typing Preservation)**.** *If $\mathcal{A}' \preceq \mathcal{A}$ and $\Lambda;\Gamma \vdash_{\mathcal{A}'} E \Longleftarrow t$ then $\Lambda;\Gamma \vdash_{\mathcal{A}} E \Longleftarrow t$.*

As $\mathbf{B}_\bot \preceq \mathcal{A}$ for any $\mathcal{A}$, the above Theorem 1.3.6 implies that abstract typing conservatively *over-approximates* concrete typing:

Corollary 1.3.7. *If* $\Lambda; \cdot \vdash E \Longleftarrow t$ *then* $\Lambda; \cdot \vdash_{\mathcal{A}} E \Longleftarrow t$.

1.4 Synthesis

Next, we formalize the concrete and abstract synthesis problems, and use the notion of abstract type checking from Sec. 1.3 to develop the TYGAR synthesis algorithm, which solves the (concrete) synthesis problem by solving a sequence of abstract synthesis problems with increasing detail.

Synthesis Problem. A *synthesis problem* (Λ, t) is a pair of a component library and *query type*. A *solution* to the synthesis problem is a normal-form term E such that $\Lambda; \cdot \vdash E :: t$. Note that the normal-form requirement does not restrict the solution space: λ_H has no higher-order functions or recursion, hence any well-typed program has an equivalent η-long β-normal form. We treat the query type as a monotype without loss of generality: any query polytype $\overline{\forall \tau}.T$ is equivalent to $[\overline{\tau \mapsto C}]T$ where $\overline{C}$ are fresh nullary type constructors. The synthesis problem in λ_H is *semi-decidable*: if a solution E exists, it can be found by enumerating programs of increasing size. Undecidability follows from a reduction from Post's Correspondence Problem (see [43]).

Abstract Synthesis Problem. An *abstract synthesis problem* $(\Lambda, t, \mathcal{A})$ is a triple of a component library, query type, and abstract cover. A *solution* to the abstract synthesis problem is a program term E such that $\Lambda; \cdot \vdash_{\mathcal{A}} E \Longleftarrow t$. We can use Corollary 1.3.7 and Theorem 1.3.5, to show that any solution to a concrete synthesis problem is also a solution to any of its abstractions:

Theorem 1.4.1. *If E is a solution to (Λ, t), then E is also a solution to $(\Lambda, t, \mathcal{A})$.*

1.4.1 Abstract Transition Nets

Next we discuss how to construct an abstract transition net (ATN) for a given abstract synthesis problem $(\Lambda, t, \mathcal{A})$, and use ATN reachability to find a solution to this synthesis problem.

Petri Nets. A *Petri net* N is a triple (P, T, E), where P is a set of places, T is a set of transitions, $E\colon (P \times T) \cup (T \times P) \to \mathbb{N}$ is a matrix of edge multiplicities (absence of an edge is represented by a zero entry). A *marking* of a Petri net is a mapping $M\colon P \to \mathbb{N}$ that assigns a non-negative number of tokens to every place. A *transition firing* is a triple $M_1 \xrightarrow{t} M_2$, such that for all places p: $M_1(p) \geq E(p,t) \wedge M_2(p) = M_1(p) - E(p,t) + E(t,p)$. A sequence of transitions $t_1, \ldots, t_n$ is a *path* between M and M' if $M \xrightarrow{t_1} M_1 \ldots M_{n-1} \xrightarrow{t_n} M'$ is a sequence of transition firings.

ATN Construction. Consider an abstract synthesis problem $(\Lambda, t, \mathcal{A})$, where $t = b_1 \to \ldots \to b_n \to b$. An *abstract transition net* $\mathcal{N}(\Lambda, t, \mathcal{A})$ is a 5-tuple (P, T, E, I, F), where (P, T, E) is a Petri net, $I\colon P \to \mathbb{N}$ is a multiset of *initial places* and $F \subseteq P$ is a set of *final places* defined as:

1. the set of *places* $P = \mathcal{A} \setminus \{\bot\}$;

2. *initial places* are abstractions of query arguments: for every $i \in [1,n]$, add 1 to $I(\alpha_{\mathcal{A}}(b_i))$;

3. *final places* are all places that subsume the query result: $F = \{A \in P \mid b \sqsubseteq A\}$.

4. for each component $c \in \Lambda$ and for each tuple $A, A_1, \ldots, A_m \in P$, where m is the arity of c, add a *transition* t to T iff $\alpha_{\mathcal{A}}(\llbracket c \rrbracket(A_1, \ldots, A_m)) \equiv A$; set $E(t,A) = 1$ and add 1 to $E(A_j,t)$ for every $j \in [1,m]$;

5. for each initial place $\{p \in P \mid I(p) > 0\}$, add a self-loop *copy transition* κ to T, setting $E(p,\kappa) = 1$ and $E(\kappa,p) = 2$, and a self-loop *delete transition* δ to T, setting $E(p,\delta) = 1$ and $E(\delta,p) = 0$.

Given an ATN $\mathcal{N} = (P, T, E, I, F)$, M_F is a *valid final marking* if it assigns exactly one token to some final place: $\exists f \in F. M_F(f) = 1 \wedge \forall p \in P. p \neq f \Rightarrow M_F(p) = 0$. A path $\pi = [t_1, \ldots, t_n]$ is a *valid path* of the ATN ($\pi \models \mathcal{N}$), if it is a path in the Petri net (P, T, E) from the marking I to some valid final marking M_F.

From Paths to Programs. Any valid path π corresponds to a set of normal-form terms $\mathsf{terms}(\pi)$. The mapping from paths to programs has been defined in prior work on SYPET, so we do not formalize it here. Intuitively, multiple programs arise because a path does not distinguish between different tokens in one place and has no notion of order of incoming edges of a transition.

```
Input: Abstract synthesis problem $(\Lambda, t, \mathcal{A})$
Output: Solution $e$ or $\bot$ if no solution

1: function SynAbstract($\Lambda, t, \mathcal{A}$)
2:     $\mathcal{N} \leftarrow \mathcal{N}(\Lambda, t, \mathcal{A})$
3:     $\pi \leftarrow$ ShortestValidPath($\mathcal{N}$)
4:     if $\pi = \bot$ then
5:         return $\bot$
6:     else
7:         for $E \in \mathsf{terms}(\pi)$ do
8:             if $\Lambda; \cdot \vdash_{\mathcal{A}} E \Longleftarrow t$ then
9:                 return $E$
```

```
Input: Synthesis problem $(\Lambda, t)$, initial cover $\mathcal{A}_0$
Output: Solution $E$ or $\bot$ if no solution

1: function Synthesize($\Lambda, t, \mathcal{A}_0$)
2:     $\mathcal{A} \leftarrow \mathcal{A}_0$
3:     while true do
4:         $E \leftarrow$ SynAbstract($\Lambda, t, \mathcal{A}$)
5:         if $E = \bot$ then
6:             return $\bot$
7:         else if $\Lambda; \cdot \vdash E \Longleftarrow t$ then
8:             return $E$
9:         else
10:            $\mathcal{A} \leftarrow$ Refine($\mathcal{A}, E, t$)
```

Figure 1.7. (left) Algorithm for the abstract synthesis problem. (right) The TYGAR algorithm.

Guarantees. ATN reachability is both sound and complete with respect to (abstract) typing:

Theorem 1.4.2 (ATN Completeness)**.** *If* $\Lambda; \cdot \vdash_{\mathcal{A}} E \Longleftarrow t$ *and* $E \in \mathsf{terms}(\pi)$ *then* $\pi \models \mathcal{N}(\Lambda, t, \mathcal{A})$.

Theorem 1.4.3 (ATN Soundness)**.** *If* $\pi \models \mathcal{N}(\Lambda, t, \mathcal{A})$*, then* $\exists E \in \mathsf{terms}(\pi)$ *s.t.* $\Lambda; \cdot \vdash_{\mathcal{A}} E \Longleftarrow t$.

Abstract Synthesis Algorithm. Fig. 1.7 (left) presents an algorithm for solving an abstract synthesis problem $(\Lambda, t, \mathcal{A})$. The algorithm first constructs the ATN $\mathcal{N}(\Lambda, t, \mathcal{A})$. Next, the function SHORTESTVALIDPATH uses a constraint solver to find a shortest valid path $\pi \models \mathcal{N}$[5]. From Theorem 1.4.2, we know that if no valid path exists (no final marking is reachable from any initial marking), then the abstract synthesis problem has no solution, so the algorithm returns $\bot$. Otherwise, it enumerates all programs $E \in \mathsf{terms}(\pi)$ and type-checks them abstractly, until it encounters an E that is abstractly well-typed (such an E must exists per Theorem 1.4.3).

ATN versus TTN. Our ATN construction is inspired by but different from the TTN construction in SYPET [27]. In the monomorphic setting of SYPET, it suffices to add a single transition per component. To account for our *polymorphic* components, we need a transition for every *abstract instance* of the component's polytype. To compute the set of abstract instances, we consider all possible *m*-tuples of places, and for each, we compute the result of the abstract type transformer

[5]Sec. 1.5.3 details our encoding of ATN reachability into constraints.

$\alpha_{\mathcal{A}}(\llbracket c \rrbracket(A_1,\ldots,A_m))$. This result is either $\perp$, in which case no transition is added, or some $A \in P$, in which case we add a transition from $A_1,\ldots,A_m$ to A.

Due to abstraction, unlike SyPet, where the final marking contains a single token in the result type b, we must allow for several possible final markings. Specifically, we allow the token to end up in any place A that subsumes b, not just in its most precise abstraction $\alpha_{\mathcal{A}}(b)$. This is because, like any abstract interpretation, abstract type inference might lose precision, and so requiring that it infer the most precise type $\alpha_{\mathcal{A}}(b)$ for the solution would lead to incompleteness.

Enforcing Relevance. Finally, consider copy transitions κ and delete transitions δ: in this section, we describe an ATN that implements a simple, structural type system, where each function argument can be used zero or more times. Hence we allow the ATN to duplicate tokens using κ transitions and discard them using δ transitions. We can easily adapt the ATN definition to implement a relevant type system by eliminating the δ transitions (this is what our implementation does, see Sec. 1.5.3); a linear type system can be supported by eliminating both.

1.4.2 The TYGAR Algorithm

The abstract synthesis algorithm from Fig. 1.7 either returns $\perp$, indicating that there is no solution to the synthesis problem, or a term E that is abstractly well-typed. However, this term may not be concretely well-typed, and hence, may not be a solution to the synthesis problem. We now turn to the core of our technique: the *type-guided abstraction refinement* (TYGAR) algorithm which iteratively refines an abstract cover $\mathcal{A}$ until it is specific enough that a solution to an abstract synthesis problem is also well-typed in the concrete type system.

Fig. 1.7 (right) describes the pseudocode for the TYGAR procedure which takes as input a (concrete) synthesis problem (Λ, t) and an initial abstract cover $\mathcal{A}_0$, and either returns a solution E to the synthesis problem or $\perp$ if t cannot be inhabited using the components in Λ. In every iteration, TYGAR first solves the abstract synthesis problem at the current level of abstraction $\mathcal{A}$, using the previously defined algorithm SYNABSTRACT. If the abstract problem has no solution, then neither does the concrete one (by Theorem 1.4.1), so the algorithm returns $\perp$. Otherwise, the algorithm type-checks the term E against the concrete query type. If it is well-typed, then E is a solution to the synthesis problem (Λ, t); otherwise E is *spurious*.

Refinement. The key step in the TYGAR algorithm is the procedure REFINE, which takes as input the current cover $\mathcal{A}$ and a spurious program E and returns a refinement $\mathcal{A}'$ of the current cover ($\mathcal{A}' \preceq \mathcal{A}$) such that E is abstractly ill-typed in $\mathcal{A}'$ ($\Lambda; \cdot \nvdash_{\mathcal{A}'} E \Longleftarrow t$). Procedure REFINE is detailed in Sec. 1.4.3, but the declarative description above suffices to see how it helps the synthesis algorithm make progress: in the next iteration, SYNABSTRACT cannot return the same spurious program E, as it no longer type-checks abstractly. Moreover, the intuition is that along with E the refinement rules out other spurious programs that are ill-typed "for a similar reason".

Initial Cover. The choice of initial cover $\mathcal{A}_0$ has no influence on the correctness of the algorithm. A natural choice is the most general cover $\mathcal{A}_\top = \{\tau, \bot\}$. In our experiments (Sec. 1.6) we found that synthesis is more efficient if we pick the initial cover $\mathcal{A}_Q(\overline{b_i} \to b) = \mathsf{close}(\{\tau, \overline{b_i}, b, \bot\})$[6], which represents the query type $t = \overline{b_i} \to b$ concretely. Intuitively, the reason is that the distinctions between the types in t are very likely to be important for solving the synthesis problem, so there is no need to make the algorithm re-discover them from scratch.

Soundness and Completeness. SYNTHESIZE is a semi-algorithm for the synthesis problem in λ_H.

Theorem 1.4.4 (Soundness)**.** *If* SYNTHESIZE$(\Lambda, t, \mathcal{A}_0)$ *returns E then* $\Lambda; \cdot \vdash E :: t$.

Proof Sketch. This follows trivially from the type check in line 7 of the algorithm. □

Theorem 1.4.5 (Completeness)**.** *If* $\exists E.\ \Lambda; \cdot \vdash E :: t$ *then* SYNTHESIZE$(\Lambda, t, \mathcal{A}_0)$ *returns some* $E' \neq \bot$.

Proof Sketch. Let E_0 be some shortest solution to (Λ, t) and let k be the number of all syntactically valid programs of the same or smaller size than E_0 (here, the size of the program is the number of component applications). Line 4 cannot return $\bot$ or a program E that is larger than

[6]Here $\mathsf{close}(\mathcal{A})$ closes the cover under meet, as required by the definition of sublattice.

```
Input: $\mathcal{A}, E, t$ s.t. $\Lambda; \cdot \nvdash E \Longleftarrow t$
Output: $\mathcal{A}' \preceq \mathcal{A}$ s.t. $\Lambda; \cdot \nvdash_{\mathcal{A}'} E \Longleftarrow t$
1: function REFINE($\mathcal{A}, \lambda \overline{x_i}.e_{body}, \overline{b_i} \to b$)
2:     $\Lambda \leftarrow \Lambda \cup (r :: b \to b)$
3:     $e^* \leftarrow r\ e_{body}$
4:     for $e_j \in \mathsf{subterms}(e^*)$ do
5:         $\Lambda; \overline{x_i : b_i} \vdash e_j \Longrightarrow U[e_j]$
6:     $U \leftarrow$ GENERALIZE($U, e^*$)
7:     return $\mathsf{close}(\mathcal{A} \cup \mathsf{range}(U))$
```

```
Input: $U, e$ s.t. $I_1 \wedge I_2 \wedge I_3$
Output: $U'$ s.t. $I_1 \wedge I_2 \wedge I_3$
1: function GENERALIZE($U, e$)
2:     if $e = x$ then
3:         return $U$
4:     else if $e = c\ \overline{e_j}$ then
5:         $\overline{B_j} \leftarrow$ weaken $\overline{U[e_j]}$ while $[\![c]\!](\overline{B_j}) \sqsubseteq U[e]$
6:         $U' \leftarrow U[\overline{e_j \mapsto B_j}]$
7:         for $e_j$ do GENERALIZE($U', e_j$)
```

Figure 1.8. Refinement algorithm.

E_0, since E_0 is abstractly well-typed at any $\mathcal{A}$ by Corollary 1.3.7, and SYNABSTRACT always returns a shortest abstractly well-typed program, when one exists by Theorem 1.4.2. Line 4 also cannot return the same solution twice by the property of REFINE. Hence the algorithm must find a solution in at most k iterations. □

When there is no solution, our algorithm might not terminate. This is unavoidable, since the synthesis problem is only semi-decidable, as we discussed at the beginning of this section. In practice, we impose an upper bound on the length of the solution, which guarantees termination.

1.4.3 Refining the Abstract Cover

This section details the refinement step of the TYGAR algorithm. The pseudocode is given in Fig. 1.8. The top-level function REFINE($\mathcal{A}, E, t$) takes as inputs an abstract cover $\mathcal{A}$, a term E, and a goal type t, such that E is ill-typed concretely ($\Lambda; \cdot \nvdash E \Longleftarrow t$), but well-typed abstractly ($\Lambda; \cdot \vdash_{\mathcal{A}} E \Longleftarrow t$). It produces a refinement of the cover $\mathcal{A}' \preceq \mathcal{A}$, such that E is ill-typed abstractly in that new cover ($\Lambda; \cdot \nvdash_{\mathcal{A}'} E \Longleftarrow t$).

Proof of untypeability. At a high-level, REFINE works by constructing a *proof of untypeability* of E, *i.e.* a mapping $U : \mathbf{e} \to \mathbf{B}_\perp$ from subterms of E to types, such that if $\mathsf{range}(U) \subseteq \mathcal{A}'$ then $\Lambda; \cdot \nvdash_{\mathcal{A}'} E \Longleftarrow t$ (in other words, the types in U contain enough information to reject E). Once U is constructed, line 7 adds its range to $\mathcal{A}$, and then closes the resulting set under meet.

Let us now explain how U is constructed. Let $E \doteq \lambda \overline{x_i}.e_{body}$, $t \doteq \overline{b_i} \to b$, and $\Gamma \doteq \overline{x_i : b_i}$.

There are two reasons why E might not type-check against t: either e_{body} on its own is ill-typed or it has a non-bottom type that nevertheless does not subsume b. To unify these two cases, REFINE constructs a new application term $e^* = r\ e_{body}$, where r is a dedicated component of type $b \to b$; such e^* is guaranteed to be ill-typed on its own: $\Lambda;\Gamma \vdash e^* \Longrightarrow \bot$. Lines 4–5 initialize U for each subterm of e^* with the result of concrete type inference. At this point U already constitutes a valid proof of untypeability, but it contains too much information; in line 6 the call to GENERALIZE removes as much information from U as possible while maintaining enough to prove that e^* is ill-typed. More precisely, GENERALIZE maintains three crucial invariants that together guarantee that U is a proof of untypeability:

I_1**:** *(U subsumes concrete typing)* For any $e \in \mathsf{subterms}(e^*)$, if $\Lambda;\Gamma \vdash e \Longrightarrow B$, then $B \sqsubseteq U[e]$;

I_2**:** *(U abstracts type transformers)* For any application subterm $e = c\ \overline{e_j}$, $[\![c]\!](\overline{U[e_j]}) \sqsubseteq U[e]$;

I_3**:** *(U proves untypeability)* $U[e^*] = \bot$.

Lemma 1.4.6. *If $I_1 \wedge I_2 \wedge I_3$ then U is a proof of untypeability: if $\mathsf{range}(U) \subseteq \mathcal{A}'$ then $\Lambda;\cdot \nvdash_{\mathcal{A}'} E \Longleftarrow t$.*

Proof Sketch. We can show by induction on the derivation that for any $\mathcal{A}' \supseteq \mathsf{range}(U)$ and node e, $\Lambda;\Gamma \vdash_{\mathcal{A}'} e \Longrightarrow B \sqsubseteq U[e]$ (base case follows from I_1, and inductive case follows from I_2). Hence, $\Lambda;\Gamma \vdash_{\mathcal{A}'} e^* \Longrightarrow B \sqsubseteq U[e^*] = \bot$ (by I_3), so $\Lambda;\Gamma \vdash_{\mathcal{A}'} e_{body} \Longrightarrow B \not\sqsubseteq b$, and $\Lambda;\cdot \nvdash_{\mathcal{A}'} E \Longleftarrow t$. $\square$

***Correctness of* GENERALIZE.** Now that we know that invariants I_1–I_3 are sufficient for correctness, let us turn to the inner workings of GENERALIZE. This function starts with the initial proof U (concrete typing), and recursively traverses the term e^* top-down. At each application node $e = c\ \overline{e_j}$ it *weakens* the argument labels $\overline{U[e_j]}$ (lines 4–7). The weakening step performs *lattice search* to find more general values for $\overline{U[e_j]}$ allowed by I_2. More concretely, each new value B_j starts out as the initial value of $\overline{U[e_j]}$; at each step, weakening picks one $B_j \neq \bot$ and moves it upward in the lattice by replacing a ground subterm of B_j with a type

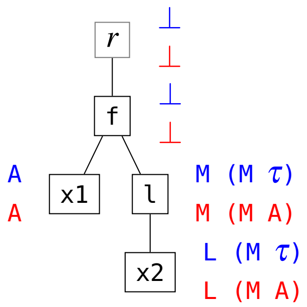


Figure 1.9. REFINE in the second iteration of the running example.

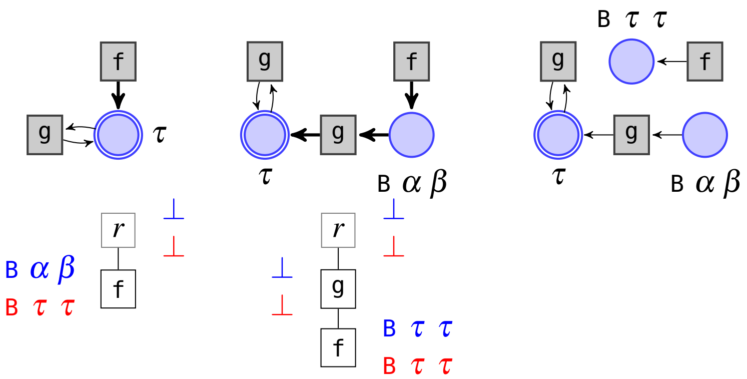


Figure 1.10. SYNTHESIZE on an unsatisfiable problem.

variable; the step is accepted as long as $[\![c]\!](\overline{B_j}) \sqsubseteq U[e]$. The search terminates when there is no more B_j that can be weakened. Note that in general there is no unique most general value for $\overline{B_j}$, we simply pick the first value we find that cannot be weakened any further. The correctness of the algorithm does not depend on the choice of $\overline{B_j}$, and only rests on two properties: (1) $\overline{U[e_j] \sqsubseteq B_j}$ and (2) $[\![c]\!](\overline{B_j}) \sqsubseteq U[e]$.

We can show that GENERALIZE maintains the invariants I_1–I_3. I_1 is maintained by property (1) of weakening (we start from concrete types and only move up in the lattice). I_2 is maintained between e and its children $\overline{e_j}$ by property (2) of weakening, and between each e_j and its children because the label of e_j only goes up. Finally, I_3 is trivially maintained since we never update $U[e^*]$.

Example 1. Let us walk through the refinement step in iteration 2 of our running example from Sec. 1.2.2. As a reminder, $\Lambda(\texttt{f}) = \forall\alpha.\alpha \to \texttt{M}\ \alpha \to \alpha$ and $\Lambda(\texttt{l}) = \forall\beta.\texttt{L}\ \beta \to \texttt{M}\ \beta$. Consider a call to REFINE$(\mathcal{A}, E, t)$, where $\mathcal{A} = \{\tau, \texttt{A}, \texttt{L}\ \tau, \perp\}$, $E = \lambda x_1\ x_2.\texttt{f}\ x_1\ (\texttt{l}\ x_2)$ and $t = \texttt{A} \to \texttt{L (M A)} \to \texttt{A}$. Let us denote $\Gamma = x_1 : \texttt{A}, x_2 : \texttt{L (M A)}$. It is easy to see that E is ill-typed concretely but well-typed abstractly, since, as explained above, $\Lambda;\Gamma \vdash_{\mathcal{A}} \texttt{l}\ x_2 \Longrightarrow \tau$, and hence $\Lambda;\Gamma \vdash_{\mathcal{A}} \texttt{f}\ x_1\ (\texttt{l}\ x_2) \Longrightarrow \texttt{A}$. REFINE first constructs $e^* = r\ e_{body}$; the AST for this term is shown on Fig. 1.9 (left). It then initializes the mapping U with concrete inferred types, depicted as red labels; as expected $U[e^*] = \perp$. The blue labels show U' obtained by calling GENERALIZE through the following series of recursive calls:

- In the initial call to GENERALIZE, the term e is $r\ e_{body}$; although it is an application, we do not weaken the label for e_{body} since its concrete type is $\bot$, which cannot be weakened.

- We move on to $e_{body} = \texttt{f}\ x_1\ l$ with $U[x_1] = \texttt{A}$ and $U[l] = \texttt{M (M A)}$. The former type cannot be weakened: an attempt to replace A with τ causes $[\![\texttt{f}]\!]$ to produce $\texttt{M A} \not\sqsubseteq \bot$. The latter type can be weakened by replacing A with τ (since $[\![\texttt{f}]\!](\texttt{A},\texttt{M (M}\ \tau\texttt{)}) = \bot$), but no further.

- The first child of f, x_1, is a variable so U remains unchanged.

- For the second child of f, $l = \texttt{l}\ x_2$, l's signature allows us to weaken $U[x_2]$ to $\texttt{L (M}\ \tau\texttt{)}$ but no further, since $[\![\texttt{l}]\!](\texttt{L (M}\ \tau\texttt{)}) = \texttt{M (M}\ \tau\texttt{)}$ but $[\![\texttt{l}]\!](\texttt{L}\ \tau) = \texttt{M}\ \tau \not\sqsubseteq \texttt{M (M}\ \tau\texttt{)}$.

- Since x_2 is a variable, GENERALIZE terminates.

Example 2. We conclude this section with an end-to-end application of TYGAR to a very small but illustrative example. Consider a library Λ with three type constructors, Z, U, and B (with arities 0, 1, and 2, respectively), and two components, f and g, such that: $\Lambda(\texttt{f}) = \forall\alpha.\texttt{B}\ \alpha\ \alpha$ and $\Lambda(\texttt{g}) = \forall\beta.\texttt{B}\ (\texttt{U}\ \beta)\ \beta \to \texttt{Z}$. Consider the synthesis problem $(\Lambda, \texttt{Z})$, which has no solutions: the only way to obtain a Z is from g, which requires a B with *distinct* parameters, but we can only construct a B with *equal* parameters (using f). Assume that the initial abstract cover is $\mathcal{A}_0 = \{\tau, \bot\}$, as shown in the upper left of Fig. 1.10. SYNABSTRACT$(\Lambda, \texttt{Z}, \mathcal{A}_0)$ returns a program f, which is spurious, hence we invoke REFINE$(\mathcal{A}_0, \texttt{f}, \texttt{Z})$. The results of concrete type inference are shown as red labels in Fig. 1.10; in particular, note that because f is a nullary component, $[\![\texttt{f}]\!]$ is simply a fresh instance of its type, here $\texttt{B}\ \tau\ \tau$, which can be generalized to $\texttt{B}\ \alpha\ \beta$: the root cause of the type error is that r does not accept a B. In the second iteration, $\mathcal{A}_0 = \{\tau, \texttt{B}\ \alpha\ \beta, \bot\}$ and SYNABSTRACT$(\Lambda, \texttt{Z}, \mathcal{A}_1)$ returns g f, which is also spurious. In this call to REFINE, however, the concrete type of f can no longer be generalized: the root cause of the type error is that g accepts a B with distinct parameters. Adding $\texttt{B}\ \tau\ \tau$ to the cover, results in the ATN on the right, which does not have a valid path (SYNABSTRACT returns $\bot$).

There are three interesting points to note about this example. (1) In general, even concrete type inference may produce non-ground types, for example: $\Lambda; \cdot \vdash$ `f` $\Longrightarrow$ `B` $\tau\ \tau$. (2) SYNTHESIZE can *sometimes* detect that there is no solution, even when the space of all possible ground base types is infinite. (3) To prove untypeability of `g f`, our abstract domain must be able to express non-linear type-level terms (*i.e.* types with repeated variables, like `B` $\tau\ \tau$); we could not, for example, replace type variables with a single construct `?`, as in gradual typing [91].

1.5 Implementation

We have implemented the TYGAR synthesis algorithm in Haskell, in a tool called H+. The tool relies on the Z3 SMT solver [22] to find paths in the ATN. This section focuses on interesting implementation details, such as desugaring Haskell libraries into first-order components accepted by TYGAR, an efficient and incremental algorithm for ATN construction, and the SMT encoding of ATN reachability.

1.5.1 Desugaring Haskell Types

The Haskell type system is significantly more expressive than that of our core language λ_H, and many of its advanced features are not supported by H+. However, two type system features are ubiquitous in Haskell: higher-order functions and type classes. As we illustrated in Sec. 1.2.4 and Sec. 1.2.5, H+ handles both features by desugaring them into λ_H. Next, we give more detail on how H+ translates a Haskell synthesis problem $(\tilde{\Lambda}, \tilde{t})$ into a λ_H synthesis problem (Λ, t):

1. Λ includes a fresh binary type constructor `F` $\alpha\ \beta$ (used to represent function types).

2. Every declaration of type class `C` τ with methods $m_i :: \forall\tau.T_i$ in $\tilde{\Lambda}$ gives rise to a type constructor `CD` τ (the dictionary type) and components $m_i :: \forall\tau.$`CD` $\tau \rightarrow T_i$ in Λ. For example, a type class declaration `class Eq` α `where (==) :: a` $\rightarrow$ `a` $\rightarrow$ `Bool` creates a fresh type constructor `EqD` α and a component `(==) :: EqD` $\alpha \rightarrow \alpha \rightarrow \alpha \rightarrow$ `Bool`.

3. Every instance declaration `C B` in $\tilde{\Lambda}$ produces a component that returns a dictionary `CD B`. So `instance Eq Int` creates a component `eqInt :: EqD Int`, while a subclass instance like `instance Eq a => Eq [a]` creates a component `eqList :: EqD a → EqD [a]`. Note that the exact implementation of the type class methods inside the instance is irrelevant; all we care about is that the instance inhabits the type class dictionary.

4. For every component c in $\tilde{\Lambda}$, we add a component c to Λ and define $\Lambda(c) = \mathsf{desugar}\left(\tilde{\Lambda}(c)\right)$, where the translation function desugar, which eliminates type class constraints and higher-order types, is defined as follows:

$$\mathsf{desugar}\,(\forall\overline{\tau}.(\texttt{C}_1\ \tau_1,\ldots,\texttt{C}_n\ \tau_n) \Rightarrow T) = \forall\overline{\tau}.\texttt{CD}_1\ \tau_1 \to \ldots \to \texttt{CD}_n\ \tau_n \to \mathsf{desugar}(T)$$

$$\mathsf{desugar}(T_1 \to T_2) = \mathsf{base}(T_1) \to \mathsf{desugar}(T_2) \qquad \mathsf{desugar}(B) = B$$

$$\mathsf{base}(T_1 \to T_2) = \texttt{F}\ \mathsf{base}(T_1)\ \mathsf{base}(T_2) \qquad \mathsf{base}(B) = B$$

For example, Haskell components on the left are translated into λ_H components on the right:

`member :: Eq α => α → [α] → Bool`	`member :: EqD α → α → [α] → Bool`
`any :: (α → Bool) → [α] → Bool`	`any :: F α Bool → [α] → Bool`

5. For every non-nullary component and type class method c in $\tilde{\Lambda}$, we add a nullary component c' to Λ and define $\Lambda(c') = \mathsf{base}(\Lambda(c))$. For example: `any' :: F (F α Bool) (F [α] Bool)`.

6. Finally, the λ_H query type t is defined as $\mathsf{desugar}(\tilde{t})$.

Limitations. Firstly, in modern Haskell, type classes often constrain *higher-kinded* type variables; for example, the `Monad` type class in the signature `return :: Monad m => a → m a` is a constraint on *type constructors* rather than *types*. Support for higher-kinded type variables is beyond the scope of this work. Secondly, in theory our encoding of higher-order functions (Sec. 1.2.4) is complete, as any program can be re-written in *point-free style*, *i.e.* without lambda terms, using an appropriate set of components [7] including an *apply* component `($) :: F` $\alpha\,\beta \to \alpha \to \beta$ that enables synthesizing terms containing partially applied functions. However, in practice we found that adding a nullary version for every component significantly increases the size of the search space and is infeasible for component libraries of nontrivial size. Hence, in our evaluation we only generate nullary variants of a selected subset of popular components.

1.5.2 ATN Construction

Incremental updates. Sec. 1.4.1 shows how to construct an ATN given an abstract synthesis problem $(\Lambda, t, \mathcal{A})$. However, computing the set of ATN transitions and edges from scratch in each refinement iteration is expensive. We observe that each iteration only makes small changes to the abstract cover, which translate to small changes in the ATN.

Let $\mathcal{A}$ be the old abstract cover and $\mathcal{A}' = \mathcal{A} \cup \{A_{new}\}$ be the new abstract cover (if a refinement step adds multiple types to $\mathcal{A}$, we can consider them one by one). Let $\mathsf{parents}$ be the direct successors of A_{new} in the $\sqsubseteq$ partial order; for example, in the abstract cover $\{\tau, \mathtt{P}\ \alpha\ \beta, \mathtt{P\ A}\ \beta, \mathtt{P}\ \alpha\ \mathtt{B}, \mathtt{P\ A\ B}, \bot\}$, the parents of $\mathtt{P\ A\ B}$ are $\{\mathtt{P\ A}\ \beta, \mathtt{P}\ \alpha\ \mathtt{B}\}$. Intuitively, adding A_{new} to the cover can *add* new transitions and *re-route* some existing transitions. A transition is re-routed if a component c returns a more precise type under $\mathcal{A}'$ than it did under $\mathcal{A}$, given the same types as arguments. Our insight is that the only candidates for re-routing are those transitions that return one of the types in $\mathsf{parents}$. Similarly, all new transitions can be derived from those that take one of the types in $\mathsf{parents}$ as an argument. More precisely, starting from the old ATN, we update its transitions T and edges E as follows:

1. Consider a transition $t \in T$ that represents the abstract instance $\alpha_{\mathcal{A}}\left(\llbracket c \rrbracket(\overline{A_i})\right) = A$ such that $A \in \mathsf{parents}$; if $\alpha_{\mathcal{A}'}\left(\llbracket c \rrbracket(\overline{A_i})\right) = A_{new}$, set $E(t, A) = 0$ and $E(t, A_{new}) = 1$.

2. Consider a transition $t \in T$ that represents the abstract instance $\alpha_{\mathcal{A}}\left(\llbracket c \rrbracket(\overline{A_i})\right) = A$ such that at least one $A_i \in \mathsf{parents}$; consider $\overline{A'_i}$ obtained from $\overline{A_i}$ by substituting at least one $A_i \in \mathsf{parents}$ with A_{new}; if $\alpha_{\mathcal{A}'}\left(\llbracket c \rrbracket(\overline{A'_i})\right) = A' \neq \bot$, add a new transition t' to T, set $E(t', A') = 1$ and add 1 to $E(A'_i, t')$ for each A'_i.

Transition coalescing. The ATN construction algorithm in Sec. 1.4.1 adds a separate transition for each abstract instance of each component in the library. Observe, however, that different components may share the same abstract instance: for example in Fig. 1.3 (1), both c and l have the type $\tau \to \tau$. Our implementation *coalesces* equivalent transitions: an optimization known in the literature as *observational equivalence reduction* [103, 2]. More precisely, we do not add a new transition if one already exists in the net with the same incoming and outgoing edges. Instead, we keep track of a mapping from each transition to a set of components. Once a valid path $[t_1, \ldots, t_n]$ is found, where each transition t_i represents a set of components, we select an arbitrary component from each set to construct the candidate program. In each refinement iteration, the transition mapping changes as follows:

1. new component instances are coalesced into new groups and added to the map, each new group is added as a new ATN transition;

2. if a component instance is re-routed, it is removed from the corresponding group;

3. transitions with empty groups are removed from the ATN.

1.5.3 SMT Encoding of ATN Reachability

Our encoding differs slightly from that in prior work on SyPet. Most notably, we use an SMT (as opposed to SAT) encoding, in particular, representing transition firings as integer variables (instead of Boolean variables). This makes our encoding more compact, which is important in our setting, since, unlike SyPet, we cannot pre-compute the constraints for a component library and use them for all queries.

ATN Encoding. Given a ATN $\mathcal{N} = (P, T, E, I, F)$, we show how to build an SMT formula ϕ that encodes all valid paths of a given length ℓ; the overall search will then proceed by iteratively increasing the length ℓ. We encode the number of tokens in each place $p \in P$ at each time step $k \in [0, \ell]$ as an integer variable $\texttt{tok}_k^p$. We encode the transition firing at each time step $k \in [0, \ell)$ as an integer variable $\texttt{fire}_k$ so that $\texttt{fire}_k = t$ indicates that the transition t is fired at time step

k. For any $x \in \{P \cup T\}$, let the *pre-image* of x be $\mathsf{pre}(x) = \{y \in P \cup T \mid E(y,x) > 0\}$ and the *post-image* of x be $\mathsf{post}(x) = \{y \in P \cup T \mid E(x,y) > 0\}$.

The formula ϕ is a conjunction of the following constraints:

1. At each time step, a valid transition is fired: $\bigwedge_{k=0}^{\ell-1} 1 \leq \mathtt{fire}_k \leq |T|$
2. If a transition t is fired at time step k then all places $p \in \mathsf{pre}(t)$ have sufficiently many tokens: $\bigwedge_{k=0}^{\ell-1} \bigwedge_{t=1}^{|T|} \mathtt{fire}_k = t \implies \bigwedge_{p \in \mathsf{pre}(t)} \mathtt{tok}_k^p \geq E(p,t)$
3. If a transition t is fired at time step k then all places $p \in \mathsf{pre}(t) \cup post(t)$ will have their markings updated at time step $k+1$: $\bigwedge_{k=0}^{\ell-1} \bigwedge_{t=1}^{|T|} \mathtt{fire}_k = t \implies \bigwedge_{p \in \mathsf{pre}(t) \cup \mathsf{post}(t)} \mathtt{tok}_{k+1}^p = \mathtt{tok}_k^p - E(p,t) + E(t,p)$
4. If none of the outgoing or incoming transitions of a place p are fired at time step k, then the marking in p does not change: $\bigwedge_{k=0}^{\ell-1} \bigwedge_{p \in P} (\bigwedge_{t \in \mathsf{pre}(p) \cup \mathsf{post}(p)} \mathtt{fire}_k \neq t) \implies \mathtt{tok}_{k+1}^p = \mathtt{tok}_k^p$
5. The initial marking is I: $\bigwedge_{p \in P} \mathtt{tok}_0^p = I(p)$.
6. The final marking is valid: $\bigvee_{f \in F} \left(\mathtt{tok}_\ell^f = 1 \wedge \bigwedge_{p \in P \setminus \{f\}} \mathtt{tok}_\ell^p = 0 \right)$.

Optimizations. Although the validity of the final marking can be encoded as in (6) above, we found that quality of solutions improves if we enumerate $f \in F$ in the order from *most to least precise*; in each iteration we enforce $\mathtt{tok}_\ell^f = 1$ (and $\mathtt{tok}_\ell^p = 0$ for $p \neq f$), and move to the next place if no solution exists. Intuitively, this works because paths that end in a more precise place lose less information, and hence are more likely to produce concretely well-typed programs.

As we mentioned in Sec. 1.4, our implementation adds copy transitions but not delete transitions to the ATN, thereby enforcing relevant typing. We have also tried an alternative encoding of relevant typing, which forgoes copy transitions, and instead allows the initial marking to contain extra tokens in initial places: $\bigwedge_{p \in \{P \mid I(p) > 0\}} \mathtt{tok}_0^p \geq I(p)$ and $\bigwedge_{p \in \{P \mid I(p) = 0\}} \mathtt{tok}_0^p = 0$. Although this alternative encoding often produces solutions faster (due to shorter paths), we found that the quality of solutions suffers. We conjecture that the original encoding works well, because it *biases* the search towards linear consumption of resources, which is common for desirable programs.

1.6 Evaluation

Next, we describe an empirical evaluation of two research questions of H+:

- **Efficiency:** Is TYGAR able to find well-typed programs quickly?
- **Quality of Solutions:** Are the synthesized code snippets interesting?

Component library. We use the same set of 291 components in all experiments. To create this set, we started with all components from 12 popular Haskell library modules,[7] and excluded seven components[8] that are highly-polymorphic yet redundant (and hence slowed down the search with no added benefit).

Query Selection. We collected 44 benchmark queries from three sources:

1. HOOGLE. We started with all queries made to HOOGLE between 1/2015 and 2/2019. Among the 3.8M raw queries, 71K were syntactically unique, and only 60K could not be exactly solved by HOOGLE. Among these, many were syntactically ill-formed (*e.g.* `FromJSON a` → `Parser a` →) or unrealizable (*e.g.* `a` → `b`). We wanted to discard such invalid queries, but had no way to identify unrealizable queries automatically. Instead we decided to reduce the number of queries by selecting only popular ones (those asked at least *five* times), leaving us with 1750 queries, and then we pruned invalid queries manually, leaving us with 180 queries. Finally, out of the 180 remaining queries, only **24** were realizable with our selected component set.

[7] `Data.Maybe`, `Data.Either`, `Data.Int`, `Data.Bool`, `Data.Tuple`, `GHC.List`, `Text.Show`, `GHC.Char`, `Data.Int`, `Data.Function`, `Data.ByteString.Lazy`, `Data.ByteString.Lazy.Builder`.

[8] `id`, `const`, `fix`, `on`, `flip`, `&`, `(.)`.

2. StackOverflow. We first collected all Haskell-related questions from StackOverflow, ranked them by their view counts, and examined the first 500. Out of 15 queries with implementations, we selected **6** that were realizable with our component set.

3. *Curated.* Since we were unable to find many API-related Haskell questions on StackOverflow, and Hoogle queries do not come with expected solutions and also tend to be easy, we supplemented the benchmark set with **17** queries from our own experience as Haskell programmers.

The resulting benchmark set can be found in Fig. 1.11.

Experiment Platform. We ran all experiments on a machine with an Intel Core i7-3770 running at 3.4Ghz with 32Gb of RAM. The platform ran Debian 10, GHC 8.4.3, and Z3 4.7.1.

1.6.1 Efficiency

Setup. To evaluate the efficiency of H+, we run it on each of the 44 queries, and report the time to synthesize the first well-typed solution that passes the demand analyzer (Sec. 1.2.3). We set the timeout to 60 seconds and take the median time over three runs to reduce the uncertainty generated by using an SMT solver. To assess the importance of Tygar, we compare five variants of H+:

1. Baseline : we *monomorphise* the component library by instantiating all type constructors with all types up to an unfolding depth of one and do not use refinement.

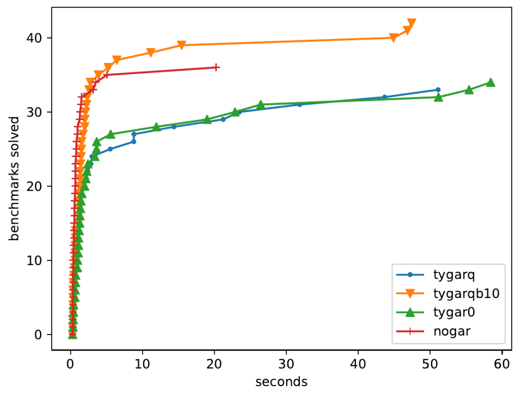
2. Nogar : we build the ATN from the abstract cover $\mathcal{A}_Q$, which precisely represents types from the query (defined in Sec. 1.4.2). We do not use refinement, and instead *enumerate* solutions to the abstract synthesis problem until one type checks concretely. Hence, this variant uses our abstract typing but does not use Tygar.

3. Tygar-*0*, which uses Tygar with the initial cover $\mathcal{A}_\top = \{\tau, \bot\}$.

4. Tygar-*Q*, which uses Tygar with the initial cover $\mathcal{A}_Q$.

N	Name	Query	Time: Total				Time: SMT Solver				Time: Type Checking			# Interesting / All		
			QB10	Q	0	NO	QB10	Q	0	NO	QB10	0	NO	H+	H-D	H-R
1	firstRight	[Either a b] -> Either a b	0.3	0.3	0.6	0.3	0.0	0.0	0.1	0.0	0.2	0.2	0.2	2/5	2/5	2/5
2	firstKey	[(a,b)] -> a	3.9	21.2	58.4		2.4	17.2	52.2		0.8	0.2		0/2	0/4	0/3
3	flatten	[[[a]]] -> [a]	1.7	5.5	1.1	0.5	0.9	2.5	0.3	0.1	0.3	0.2	0.4	5/5	5/5	0/5
4	repl-funcs	(a->b)->Int->[a->b]	0.4	0.4	0.7	0.5	0.0	0.0	0.1	0.0	0.3	0.3	0.4	2/5	2/5	1/5
5	containsEdge	[Int] -> (Int,Int) -> Bool	15.4	14.4	19.0	5.1	13.2	12.1	15.9	0.8	1.8	0.4	4.1	0/5	0/5	0/5
6	multiApp	(a -> b -> c) -> (a -> b) -> a -> c	1.2	2.4	1.2	0.5	0.4	0.9	0.5	0.2	0.3	0.2	0.2	1/5	1/5	1/5
7	appendN	Int -> [a] -> [a]	0.3	0.3	0.3	0.3	0.0	0.0	0.0	0.0	0.2	0.3	0.2	2/5	2/5	0/5
8	pipe	[(a -> a)] -> (a -> a)	0.7	0.6	2.1	0.7	0.1	0.1	0.6	0.1	0.2	0.7	0.6	1/5	1/5	0/5
9	intToBS	Int64 -> ByteString	0.6	0.6	1.6	0.3	0.1	0.1	0.5	0.0	0.3	0.3	0.2	3/5	3/5	0/5
10	cartProduct	[a] -> [b] -> [[(a,b)]]	1.5	8.8	1.3	1.3	0.6	5.5	0.4	0.5	0.3	0.2	0.6	0/5	0/5	0/5
11	applyNtimes	(a->a) -> a -> Int -> a	6.4	23.5	0.6	1.0	4.9	19.8	0.2	0.3	1.2	0.3	0.6	0/5	0/5	0/5
12	firstMatch	[a] -> (a -> Bool) -> a	1.5	1.4	2.4	0.5	0.7	0.6	1.3	0.2	0.2	0.2	0.3	5/5	5/5	5/5
13	mbElem	Eq a => a -> [a] -> Maybe a	46.8		5.6		45.5		4.0		0.8	0.3		0/3	0/3	0/5
14	mapEither	(a -> Either b c) -> [a] -> ([b], [c])	2.6	43.7	55.4	3.5	1.7	37.6	49.8	0.5	0.3	0.2	1.7	1/4	1/5	1/1
15	hoogle01	(a -> b) -> [a] -> b	0.5	0.5	1.1	0.3	0.1	0.1	0.3	0.0	0.3	0.3	0.2	2/5	2/5	2/5
16	zipWithResult	(a->b)->[a]->[(a,b)]	11.1				9.2				0.7			1/2	1/2	0/5
17	splitStr	String -> Char -> [String]	0.7	0.7	1.0	0.4	0.2	0.1	0.3	0.1	0.3	0.3	0.2	0/5	0/5	0/5
18	lookup	Eq a => [(a,b)] -> a -> b	0.7	0.7	0.7	0.8	0.2	0.2	0.2	0.3	0.3	0.3	0.3	1/5	1/3	1/4
19	fromFirstMaybes	a -> [Maybe a] -> a	1.4	3.0	3.4	0.7	0.3	0.9	1.2	0.1	0.7	0.8	0.5	2/5	2/5	0/5
20	map	(a->b)->[a]->[b]	0.3	0.3	0.4	0.4	0.0	0.0	0.1	0.0	0.2	0.2	0.3	5/5	5/5	0/5
21	maybe	Maybe a -> a -> Maybe a	0.3	0.4	0.4	0.6	0.1	0.0	0.1	0.1	0.2	0.2	0.5	2/5	1/5	0/5
22	rights	[Either a b] -> Either a [b]	1.5	31.9	11.9	0.8	0.6	20.4	5.7	0.1	0.4	0.3	0.6	1/2	1/2	1/5
23	mbAppFirst	b -> (a -> b) -> [a] -> b	2.0	1.3	2.0	0.4	1.2	0.4	0.9	0.1	0.3	0.3	0.3	1/3	1/5	0/5
24	mergeEither	Either a (Either a b) -> Either a b	2.8			1.0	1.7			0.1	0.6		0.7	0/3	0/3	0/5
25	test	Bool -> a -> Maybe a	1.4	8.8	26.4	0.7	0.7	7.1	24.3	0.3	0.2	0.3	0.3	2/5	2/5	0/5
26	multiAppPair	(a -> b, a -> c) -> a -> (b, c)	2.0			1.5	1.2			0.3	0.5		1.0	1/2	1/4	0/5
27	splitAtFirst	a -> [a] -> ([a], [a])	0.6	0.6	2.3	0.4	0.1	0.1	1.1	0.1	0.3	0.3	0.2	2/5	2/5	0/5
28	2partApp	(a->b)->(b->c)->[a]->[c]	2.3	2.2	22.9	1.5	1.2	1.2	18.7	0.5	0.2	0.3	0.3	1/5	1/5	0/5
29	areEq	Eq a => a -> a -> Maybe a	44.9				40.3				3.8			0/2	0/5	0/5
30	eitherTriple	Either a b -> Either a b -> Either a b	5.3			3.2	1.9			0.1	2.8		2.9	0/5	0/5	0/5
31	mapMaybes	(a -> Maybe b) -> [a] -> Maybe b	0.5	0.5	1.1	0.3	0.1	0.1	0.3	0.0	0.3	0.2	0.2	2/5	2/5	2/5
32	head-rest	[a] -> (a, [a])	1.4	51.1	1.0	0.8	0.7	40.6	0.3	0.1	0.2	0.3	0.6	3/5	3/5	2/5
33	appBoth	(a -> b) -> (a -> c) -> a -> (b, c)	2.1	2.8	51.1		1.3	1.5	44.3		0.3	0.3		1/5	1/5	1/1
34	applyPair	(a -> b, a) -> b	1.2	1.1	3.6	0.6	0.4	0.4	1.6	0.1	0.2	0.3	0.4	2/3	2/5	1/5
35	resolveEither	Either a b -> (a->b) -> b	1.0	1.3	1.5	0.5	0.4	0.5	0.6	0.2	0.2	0.2	0.2	1/5	1/2	1/5
36	head-tail	[a] -> (a,a)	2.2			20.2	1.5			0.4	0.3		18.8	0/5	0/5	0/5
37	indexesOf	([(a,Int)] -> [(a,Int)]) -> [a] -> [Int] -> [Int]														
38	app3	(a -> b -> c -> d) -> a -> c -> b -> d	0.3	0.3	0.3	0.3	0.0	0.0	0.0	0.0	0.2	0.3	0.2	1/5	1/5	1/5
39	both	(a -> b) -> (a, a) -> (b, b)	1.1			1.3	0.5			0.2	0.3		1.0	1/1	1/1	0/5
40	takeNdropM	Int -> Int -> [a] -> ([a], [a])	0.4	0.4	1.3	0.4	0.0	0.0	0.4	0.0	0.3	0.3	0.3	5/5	5/5	0/5
41	firstMaybe	[Maybe a] -> a	1.2	1.6	1.4	0.7	0.5	0.6	0.4	0.1	0.2	0.2	0.5	4/5	4/5	2/5
42	mbToEither	Maybe a -> b -> Either a b	47.4				21.7				24.2			0/2	0/5	0/5
43	pred-match	[a] -> (a -> Bool) -> Int	1.1	1.1	3.6	0.4	0.4	0.4	2.0	0.1	0.3	0.3	0.2	3/5	3/5	3/5
44	singleList	Int -> [Int]	0.3	0.3	0.4	0.3	0.0	0.0	0.0	0.0	0.2	0.2	0.3	1/5	1/5	0/5

Figure 1.11. H+ synthesis times and solution quality on 44 benchmarks. We report the total time to first solution, time spend in the SMT solver, and time spent type checking (including demand analysis). 'QB10', 'Q', '0', 'NO' correspond to four variants of the search algorithm: TYGAR-*QB* [10], TYGAR-*Q*, TYGAR-*0*, and NOGAR . All times are in seconds. Absence indicates no solution found within the timeout of 60 seconds. Last three columns report the number of interesting solutions among the first five (or fewer, if fewer solutions were found within the timeout of 100 seconds). 'H+', 'H-D', and 'H-R' correspond, respectively, to the default configuration of H+, disabling the demand analyzer, and using structural typing over relevant typing.



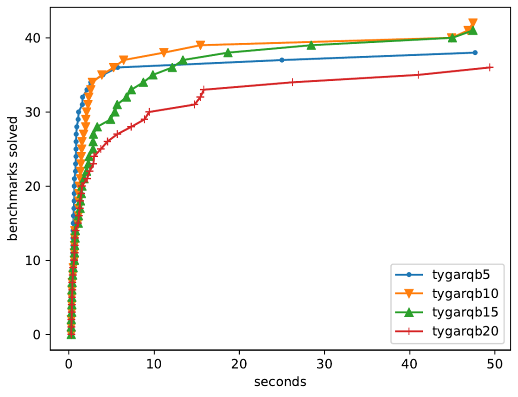


Figure 1.12. Queries solved over time for our initial variants as well as the best refinement bound.

Figure 1.13. Queries solved over time for varying refinement bounds. The variant's number indicates the boundary of refinement steps.

5. TYGAR-*QB* [*N*], which is like TYGAR-*Q*, but the size of the abstract cover is *bounded*: once the cover reaches size *N*, it stops further refinement and reverts to NOGAR -style enumeration.

Results. Fig. 1.11 reports total synthesis time for four out of the five variants. BASELINE did not complete any benchmark within 60 seconds: it spent all this time creating the TTN, and is thus is omitted from tables and graphs. Fig. 1.12 plots the number of successfully completed benchmarks against time taken for the remaining four variants (higher and weighted to the left is better). As you can see, NOGAR is quite fast on easy problems, but then it plateaus, and can only solve 37 out of 44 queries. On the other hand, TYGAR-*0* and TYGAR-*Q* are slower, and only manage to solve 35 and 34 queries, respectively. After several refinement iterations, the ATNs grow too large, and these two variants spend a lot of time in the SMT solver, as shown in columns *st-Q* and *st-0* in Fig. 1.11. Other than BASELINE , no other variant spent any meaningful amount of time building the ATN.

Bounded Refinement. We observe that NOGAR and TYGAR-*Q* have complimentary strengths and weaknesses: although NOGAR is usually faster, TYGAR-*Q* was able to find some solutions that NOGAR could not (for example, query 33: `appBoth`). We conclude that refinement is able to discover interesting abstractions, but because it is forced to make a new distinction between types in every iteration, after a while it is bound to start making irrelevant distinctions, and the ATN grows too large for the solver to efficiently navigate. To combine the strengths of the two approaches, we consider TYGAR-*QB*, which first uses refinement, and then switches to enumeration once the ATN reaches a certain bound on its number of places. To determine the optimal bound, we run the experiment with bounds 5, 10, 15, and 20.

Fig. 1.13 plots the results. As you can see, for easy queries, a bound of 5 performs the best: this correspond to our intuition that when the solution is easily reachable, it is faster to simply enumerate more candidates than spend time on refinement. However, as benchmarks get harder, having more places at ones disposal renders searches faster: the bounds of 10 and 15 seem to offer a sweet spot. Our best variant—TYGAR-*QB* [10]—solves 43 out of 44 queries with the median synthesis time of 1.4 seconds; in the rest of this section we use TYGAR-*QB* [10] as the default H+ configuration.

TYGAR-*QB* [10] solves all queries that were solved by NOGAR plus six additional queries on which NOGAR times out. A closer look at these six queries indicates that they tend to be more complex. For example, recall that NOGAR times out on the query `appBoth`, while TYGAR-*QB* [10] finds a solution of size four in two seconds. Generally, our benchmark set is favorable for NOGAR : most HOOGLE queries are easy, both because of programmers' expectations of what HOOGLE can do and also because we do not know the desired solution, and hence consider any (relevantly) well-typed solution correct. The benefits of refinement are more pronounced on queries with solution size four and higher: TYGAR-*QB* [10] solves 6 out of 7, while NOGAR solves only 2.

1.6.2 Quality of Solutions

Setup. To evaluate the quality of the solutions, we ask H+ to return, for each query, at most *five* well-typed results within a timeout of 100 seconds. Complete results are available in [43]. We then manually inspect the solutions and for each one determine whether it is *interesting*, *i.e.* whether it is something a programmer might find useful, based on our own experience as Haskell programmers[9]. Fig. 1.11 reports for each query, the *number* of interesting solutions, divided by the number of total solutions found within the timeout. To evaluate the effects of relevant typing and demand analysis (Sec. 1.2.3), we compare three variants of H+: (1) H+ with all features enabled, based on TYGAR-*QB*, labeled `H+`. (2) Our tool without the demand analyzer filter, labeled `H-D`. (3) Our tool with structural typing in place of the relevant typing, labeled `H-R` (in this variant, the SMT solver is free to choose any non-negative number of tokens to assign to each query argument).

Analysis. First of all, we observe that whenever an interesting solution was found by `H-D` or `H-R`, it was also found by `H+`, indicating that our filters are not overly conservative. We also observe that on easy queries—taking less than a second—demand analysis and relevant typing did little to help: if an interesting solution were found, then all three variants would find it and give it a high rank. However, on medium and hard queries—taking longer than a second—the demand analyzer and relevant typing helped promote interesting solutions higher in rank. Overall, 66/179 solutions produces by `H+` were interesting (37%), compared with 65/189 for `H-D` (34%) and 26/199 for `H-R` (13%). As you can see, relevant typing is essential to ensure that interesting solutions even get to the top five, whereas demand analysis is more useful to reduce the total number of solutions the programmer has to sift through. This is not surprising, since relevant typing mainly filters out *short* programs while demand analysis is left to deal with *longer* ones. In our experience, demand analysis was most useful when queries involved types like `Either a b`, where one could produce a value of type `a` from a value of type `b` by constructing and destructing the `Either`. One final observation is that in benchmarks 14, 18, 33, and 35, `H-R` found fewer results *in total* that the other two versions; we attribute this to the SMT solver struggling with determining the appropriate token multiplicities for the initial marking.

Noteworthy solutions. We presented three illustrative solutions generated by H+ as examples

[9]Unfortunately, we do not have ground truth solutions for most of our queries, so we have to resort to subjective analysis.

throughout Sec. 1.2:

- `a → [Maybe a] → a` corresponds to benchmark 19 (`fromFirstMaybes`); the solution from Sec. 1.2 is generated at rank 18.
- `(a → a) → a → Int → a` corresponds to benchmark 11 (`applyNTimes`); the solution from Sec. 1.2 is generated at rank 10.
- `Eq a => [(a,b)] → a → b` corresponds to benchmark 18 (`lookup`); the solution from Sec. 1.2 is generated at rank 1.

H+ has also produced code snippets that surprised us, and one example of them is the query `(a → b, a) → b`. On this query, the authors' intuition was to destruct the pair then apply the function. Instead H+ produces `\x → uncurry ($) x` or alternatively `\x → uncurry id x`, both of which, contrary to our intuition, are not only well-typed, but also are functionally equivalent to our intended solution. It was welcome to see a synthesis tool write more succinct code that its authors.

1.7 Related Work

Finally, we situate our work with other research into ways of synthesizing code that meets a given specification. For brevity, we restrict ourselves to the (considerable) literature that focuses on using *types* as specifications, and omit discussing methods that use *e.g.* input-output examples or tests [39, 55, 62, 75], logical specifications [95, 31] or program sketches [94].

API Search. Modern IDEs support various forms of code-completion, based on at the very least common prefixes of names (*e.g.* completing `In` into `Integer` or `fo` into `foldl'`) and so on. Many tools use type information to only return completions that are well-typed at the point of completion. This approach is generalized by search based tools like HOOGLE [68] that search for type isomorphisms [24] to find functions that "match" a given type signature (query). The above can be viewed as returning single-component results, as opposed to our goal of searching for terms that *combine* components in order to satisfy a given type query.

Search using Statistical Models. Several groups have looked into using statistical methods to improve search-based code-completion. One approach is to analyze large code bases to precompute statistical models that can be used to predict the *most likely* sequences of method calls at a given point or that yield values of a given (first order) type [86]. It is possible to generalize the above to train probabilistic models (grammars) that *generate* the most likely programs that must contain certain properties like method names, types, or keywords [69]. We conjecture that while the above methods are very useful for effectively searching for commonly occurring code snippets, they are less useful in functional languages, where higher-order components offer high degree of compositionality and lead to less code repetition.

Type Inhabitation. The work most directly related to ours are methods based on finding terms that *inhabit* a (query) type [98]. One approach is to use the correspondence between types and logics, to reduce the inhabitation question to that of validity of a logical formula (encoding the type). A classic example is DJINN [5] which implements a decision procedure for intuitionistic propositional calculus [25] to synthesize terms that have a given type. Recent work by Rehof *et al.* extends the notion of inhabitation to support object oriented frameworks whose components behaviors can be specified via intersection types [48]. However, both these approaches lack a *relevancy* requirement of its snippets, and hence return undesirable results. For example, when queried with a type `a` $\rightarrow$ `[a]`, DJINN would yield a function that always returns the empty list. One way to avoid undesirable results is to use dependent or refinement types to capture the semantics of the desired terms more precisely. SYNQUID [81] and MYTH2 [30] use different flavors of refinement types to synthesize recursive functions, while AGDA [74] makes heavy use of proof search to enable type- or hole-driven development. However, unlike H+, methods based on classical proof search do not scale up to large component libraries.

Scalable Proof Search. One way to scale search is explored by [77] which uses a very restricted form of inhabitation queries to synthesize local "auto-completion" terms corresponding to method names, parameters, field lookups and so on, but over massive component libraries (*e.g.* the .NET framework). In contrast, the INSYNTH system [47] addresses the problem of scalability by extending proof search with a notion of *succinctness* that collapses types into equivalence classes, thereby abstracting the space over which proof search must be performed. Further, INSYNTH uses *weights* derived from empirical analysis of library usage to bias the search to more likely results. However, INSYNTH is limited to simple types *i.e.* does not support parametric polymorphism which is the focus of our work.

Graph Reachability. Our approach is directly inspired by methods that reduce the synthesis problem to some form of *reachability*. PROSPECTOR [64] is an early exemplar where the components are *unary* functions that take a single input. Consequently, the component library can be represented as a directed graph of edges between input and output types, and synthesis is reduced to finding a path from the query's input type to its output type. SYPET [27], which forms the basis of our work, is a generalization of PROSPECTOR to account for general first-order functions which can take multiple inputs, thereby generalizing synthesis to reachability on Petri nets. The key contribution of our work is the notion of TYGAR that generalizes SYPET's approach to polymorphic and higher-order components.

Counterexample-Guided Abstraction Refinement. While the notion of counterexample-guided abstraction refinement (CEGAR) is classical at this point [17], there are two lines of work in particular closely related to ours. First, [32, 56] describe an iterative abstraction-refinement process for verifying Petri nets, using SMT [26]. However, in their setting, the refinement loop is used to perform unbounded verification of the (infinite-state) Petri net. In contrast, H+ performs a bounded search on each Petri net, but uses TYGAR to refine the net itself with new type instantiations that eliminate the construction of ill-typed terms. Second, BLAZE [103] describes a CEGAR approach for synthesizing programs from input-output examples, by iteratively refining *finite tree-automata* whose states correspond to values in a predicate-abstraction domain.

Programs that do not satisfy the input-output tests are used to iteratively refine the domain until a suitable correct program is found. Our approach differs in that we aim to synthesize terms of a given *type*. Consequently, although our refinement mechanism is inspired by BLAZE, we develop a novel abstract domain—a finite sub-lattice of the type subsumption lattice—and show how to use proofs of *untypeability* to refine this domain. Moreover, we show how CEGAR can be combined with Petri nets (as opposed to tree automata) in order to enforce relevancy.

Types and Abstract Interpretation. The connection between types and abstract interpretation (AI) was first introduced in [18]. The goal of their work, however, was to cast *existing* type systems in the framework of AI, while we use this framework to systematically construct *new* type systems that further abstract an existing one. More recently, [33] used the AI framework to formalize *gradual typing*. Like that work, we use AI to derive an abstract type system for our language, but otherwise the goals of the two techniques are very different. Moreover, as we hint in Sec. 1.4.3, our abstract domain is subtly but crucially different from traditional gradual typing, because our refinement algorithm relies on non-linear terms (*i.e.* types with repeated variables).

1.8 Acknowledgements

This chapter, in part, is a reprint of the material as it appears in Proceedings of the ACM on Programming Languages. Volume 4 (POPL), Article 12. Zheng Guo, Michael B. James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, Nadia Polikarpova. ACM 2020. The dissertation author was a principal author and investigator of this paper.

Chapter 2

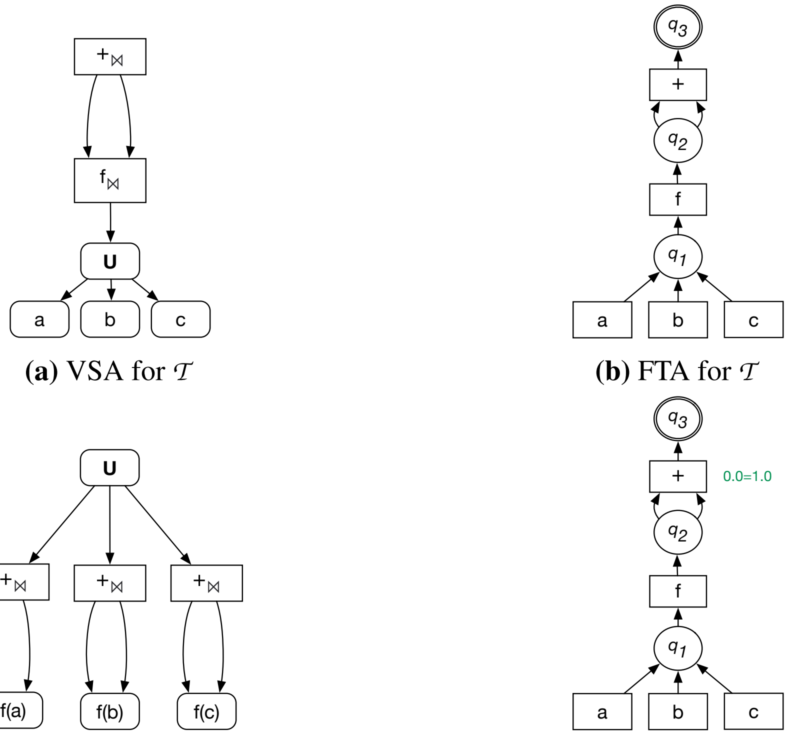
Type-Directed Program Synthesis Using Equality-Constrained Tree Automata

2.1 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) [61, 83], the data structure behind the successful spreadsheet-by-example tool FLASHFILL [40]. 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. 2.1a 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 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 [23], rewrite-based optimization [97], domain-specific synthesis [70, 71], and semantic code search [85]. Both VSAs and e-graphs



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

Figure 2.1. Representations of $\mathcal{T} = \{\mathsf{f}(t_1) + \mathsf{f}(t_2)\}$ and $\mathcal{U} = \{\mathsf{f}(t) + \mathsf{f}(t)\}$, where $t, t_1, t_2 \in \{\mathsf{a}, \mathsf{b}, \mathsf{c}\}$.

are now known [82, 58] to be equivalent to special cases of *finite tree automata* (FTAs), which have independently experienced a surge of interest in recent years [1, 102, 104]. Fig. 2.1b shows an FTA that represents the same term space as the VSA in Fig. 2.1a. 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

[1]We omit e-graphs from Fig. 2.1 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 Sec. 2.9.

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 [68, 35], we might want to represent the space of all *types* of library functions that unify with a given query type, such as $\texttt{List}\ \alpha \rightarrow \texttt{List}\ \alpha$.

Existing data structures are incapable of fully exploiting shared structure in such entangled spaces. Fig. 2.1c 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 [61] 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. 2.1d shows an ECTA that represents the term space $\mathcal{U}$. It is identical to the FTA in Fig. 2.1b 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. 2.1c.

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 Sec. 2.7.1, 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

[2]This might remind some readers of Dauchet's *reduction automata*; we postpone a detailed comparison to related work (Sec. 2.9).

inspired by constraint-based type inference. Instead of making an *eager* choice at a constrained state, such as q_1 in Fig. 2.1d, 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 Sec. 2.2, 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 Sec. 2.2 of using static reduction to "prune" away states that cannot be part of any term that satisfies the constraints.

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

1. We introduce the *ECTA data structure* (Sec. 2.3), which supports compact representation of program spaces with dependent joins, as well as efficient enumeration (Sec. 2.4) 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 (Sec. 2.5).

2. We develop *ECTA encodings* for two diverse domains: Boolean satisfiability and type-driven program synthesis (Sec. 2.7). 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 (Sec. 2.8).

The experiments show that our ECTA-based synthesizer HECTARE significantly outperforms its state-of-the-art competitor HOOGLE+ [44], 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 7× *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.

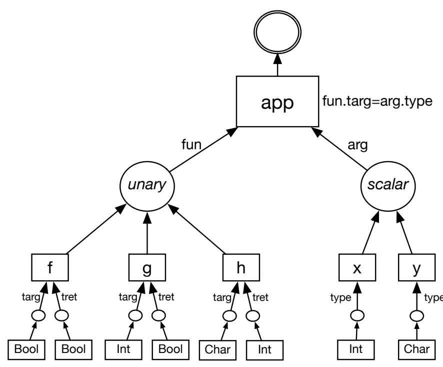
2.2 ECTA by Example

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 *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 Sec. 2.7, along an with ECTA encoding of another problem domain.

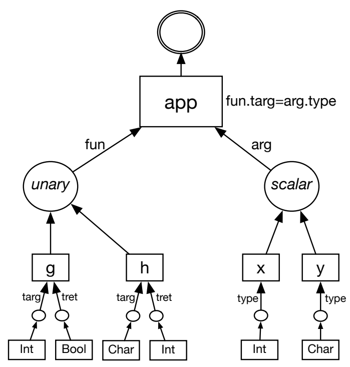
2.2.1 Representing Spaces of Well-Typed Terms

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. 2.2a.

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



(a) Initial ECTA.



(b) Reduced ECTA.

Figure 2.2. ECTAs representing all well-typed size-two terms in the environment Γ_1.

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.)

2.2.2 Static Reduction

How would one go about enumerating the terms represented by the ECTA in Fig. 2.2a? 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

[3]In our full encoding in Sec. 2.7 we remove the distinction between the terms of different arity in order to support higher-order and partial applications.

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).

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. 2.2b. 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. 2.2a, 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.

2.2.3 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. 2.3 (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. 2.2b 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

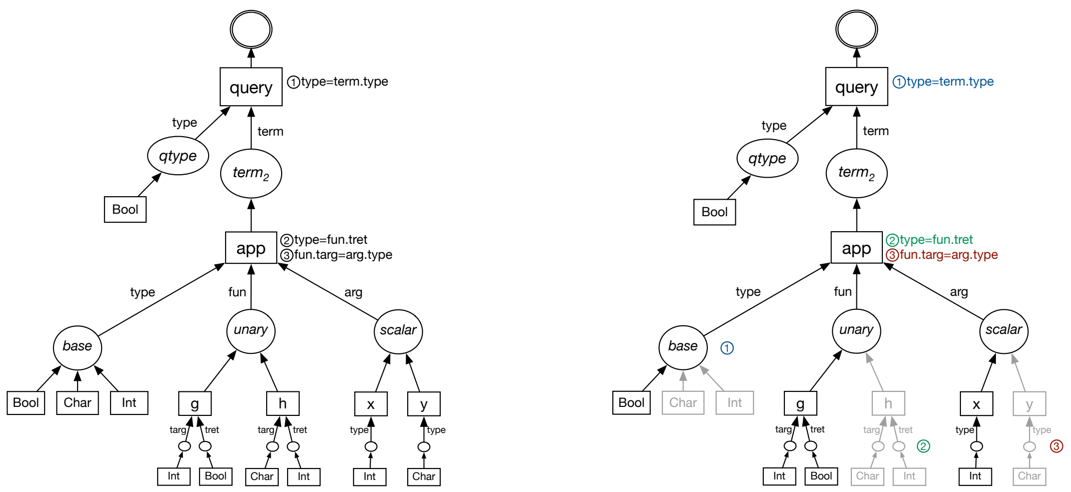


Figure 2.3. ECTAs representing size-two terms of type `Bool`. The diagram on the right shows how a sequence of static reductions on the constraints ①, ②, and ③ eliminates the grayed-out transitions and nodes.

undetermined (can be any base type), but is restricted by a new constraint ② to coincide with the return type of the function.

Fig. 2.3 (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, 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.

2.2.4 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. 2.4, which is similar to the one in Fig. 2.3. The only interesting difference is how the polymorphic type of g is represented: the type variable α is encoded as a

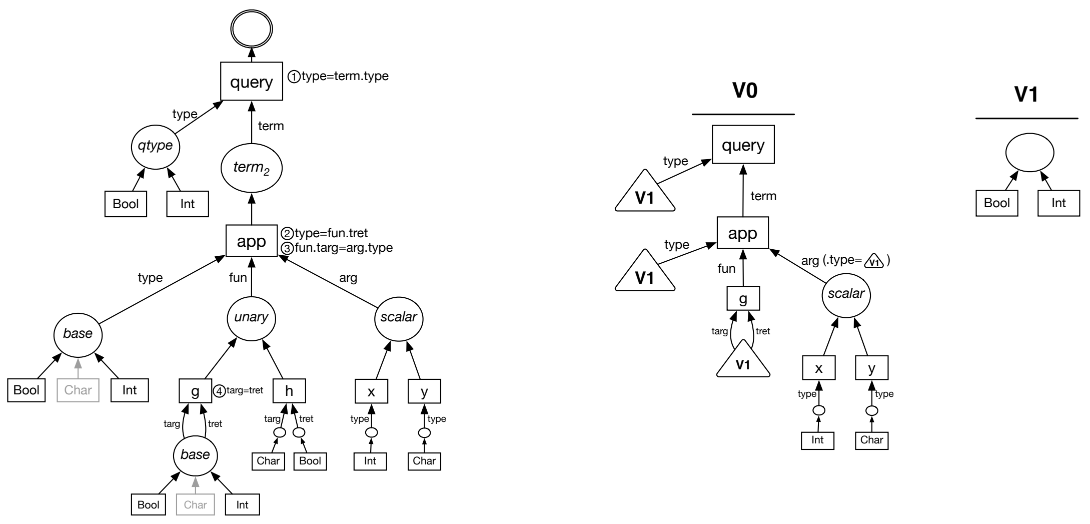


Figure 2.4. Type-driven synthesis with polymorphic functions. Left: ECTA representing all terms of types `Bool` or `Int` in Γ_2. Grayed-out transitions are eliminated by static reduction. Right: Intermediate state during enumeration. The choice of the query type has been suspended into an auxiliary automaton V_1.

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 `Bool` $\rightarrow$ `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. 2.4 (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

[4]Here we consider a limited form of polymorphism, where type variables can be instantiated only with base types; this restriction is relaxed in Sec. 2.7.

the branches. Importantly, 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. 2.4 (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 $\{\texttt{Int}\}$ *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.

2.3 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 [59].

2.3.1 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

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

$s(t_0,\ldots,t_{k-1})$ where $s \in \Sigma$, $k = \mathsf{arity}(s)$, 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) $t|_\varepsilon = t$ (ii) $s(t_0,\ldots,t_{k-1})|_{i.p} = t_i|_p$ if $i < k$ and $\perp$ otherwise. For example, for $t = +(f(a), f(b))$: $t|_{0.0} = a$, $t|_{1.0} = b$ and $t|_{2.0} = \perp$.

2.3.2 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. 2.1d. 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 equivalence classes* (PECs).

Definition 2.3.1 (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:

Definition 2.3.2 (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 2.3.3 (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 (1), we first declaratively

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

define the *closure* operation on PCSs, and then discuss how to implement it efficiently.

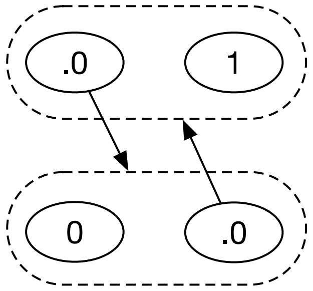
Definition 2.3.4 (Closure)**.** *A PCS C is* closed *if the following holds for any $c_1, c_2 \in C$: for any paths p, p', p'', if $p', p'' \in c_1$ and $p'.p \in c_2$, then $p''.p \in c_2$. In other words, whenever c_2 contains an extension of a path in c_1, it also contains the same extension of* all *paths in c_1. The* closure *of C, denoted $\mathsf{cl}(C)$, 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 $\mathsf{cl}(C) = \{c'_1, c'_2\}$, 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 2.3.5 (Correctness of Closure)**.** *For any term $t \in \mathcal{T}(\Sigma)$, $t \models C \Leftrightarrow t \models \mathsf{cl}(C)$.*

Theorem 2.3.6 (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 2.3.5 and Theorem 2.3.6 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 [72]. This algorithm finitely 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 $\{\{0 = 1.0\}, \{0.0 = 1\}\}$.



Syntax

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

Denotation

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

Figure 2.5. Acyclic ECTAs: syntax and semantics. Here $s \in \Sigma$ and p is a path.

2.3.3 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 [83].

Syntax. Fig. 2.5 (top) 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 $\mathsf{arity}(s)$; 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, $\mathtt{U}()$, as $\mathtt{U}_\perp$. A *normalized* ECTA contains no occurrences of $\mathbf{\Pi}_\perp$or $\mathtt{U}_\perp$, unless the root is itself $\mathtt{U}_\perp$. Any ECTA can be normalized by iteratively replacing any transition containing a $\mathtt{U}_\perp$child with $\mathbf{\Pi}_\perp$, and removing all instances of $\mathbf{\Pi}_\perp$from the children of each node. For instance, $\mathtt{U}(\mathbf{\Pi}(\mathtt{a}), \mathbf{\Pi}(+, [\mathtt{U}(\mathbf{\Pi}(\mathtt{b})), \mathtt{U}_\perp]))$

[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.

normalizes to $\mathtt{U}(\mathbf{\Pi}(\mathsf{a}))$. 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. 2.5 (bottom) 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, $\mathsf{sk}(n)$, 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: $t \notin [\![n]\!]^N \wedge t \in [\![\mathsf{sk}(n)]\!]^N$.

2.3.4 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 2.3.7 (Union)**.** *Let* $n_1 = \mathtt{U}(\overline{e_1}), n_2 = \mathtt{U}(\overline{e_2})$ *be two nodes. Then* $n_1 \sqcup n_2 = \mathtt{U}(\overline{e_1} \cup \overline{e_2})$.

Fig. 2.6 gives an example of ECTA union $n_u = n_1 \sqcup n_2$.

Theorem 2.3.8 (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 2.3.9 (Intersection)**.** *Let* $n_1 = \mathtt{U}(\overline{e_1}), n_2 = \mathtt{U}(\overline{e_2})$ *be two nodes, then:*

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

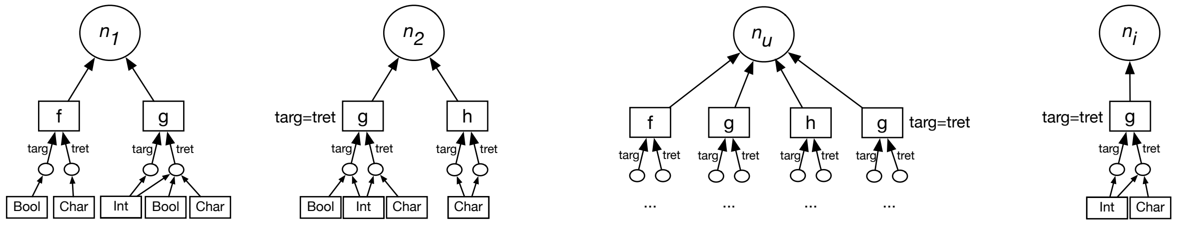


Figure 2.6. Two ECTAs n_1 and n_2, their union n_u and intersection n_i.

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:*

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

Consider the example of ECTA intersection $n_i = n_1 \sqcap n_2$ in Fig. 2.6. 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 2.3.10 (Correctness of ECTA Intersection)**.** $[\![n_1 \sqcap n_2]\!]^N = [\![n_1]\!]^N \cap [\![n_2]\!]^N$.

Proposition 2.3.11. $\mathtt{U}_\perp \sqcap n = n \sqcap \mathtt{U}_\perp = \mathtt{U}_\perp$

Corollary 2.3.12. *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* $\mathtt{U}_\perp$ *as the bottom element, and* $(\prec)$ *as the order.*

2.3.5 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. 2.2. Intuitively, the constraint `fun.targ = arg.type` has been *reduced* in Fig. 2.2b, 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 2.3.13 (Nodes at path, Subautomaton at a path)**.** *The set* $\mathsf{nodes}(n,p)$ *of nodes reachable from* $n = \mathtt{U}(\overline{e})$ *via path* p *is defined as:*

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

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

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

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. 2.2a, if n is the root node, then $\mathsf{nodes}(n,\mathtt{arg.type}) = \{\mathtt{U}(\mathbf{\Pi}(\mathtt{Int})),\mathtt{U}(\mathbf{\Pi}(\mathtt{Char}))\}$ and $n|_{\mathtt{arg.type}} = \mathtt{U}([\mathbf{\Pi}(\mathtt{Int}),\mathbf{\Pi}(\mathtt{Char})])$. 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 Sec. 2.7.2.

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 2.3.14 (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 2.3.20).

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

Definition 2.3.15 (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 $n \in$ `nodes`(e, p_i), $n \sqcap e|_{p_j} \neq \mathtt{U}_\perp$.*

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 `nodes`(e, p_1) will match some node in `nodes`(e, p_2). For example, to reduce the constraint `fun.targ = arg.type` at the transition `app` in Fig. 2.2, the algorithm first computes $\mathtt{app}|_{\mathtt{arg.type}}$, the automaton representing all possible actual parameter types; the result is $n_a = \mathtt{U}(\mathbf{\Pi}(\mathtt{Int}), \mathbf{\Pi}(\mathtt{Char}))$. 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 $\mathtt{U}_\perp$, leading to the removal of the f transition upon normalizatoin and resulting in Fig. 2.2b.

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

Definition 2.3.16 (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 $n = \mathtt{U}(\overline{e})$:*

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

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

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

For example, in Fig. 2.2a, intersecting the root node n at path `fun.targ` with the node n_a from our previous example ($n_a = \mathtt{U}(\mathbf{\Pi}(\mathtt{Int}), \mathbf{\Pi}(\mathtt{Char}))$) yields the ECTA in Fig. 2.2b.

Lemma 2.3.17. *$t \in [\![\left(n|^{\sqcap n'}_{p}\right) |_p]\!]^N$ 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. 2.2 gets reduced: once we have obtained the "actual parameter automaton" $n_a = \mathsf{app}|_{\mathtt{arg.type}}$, we can simply return $n|^{\sqcap n_a}_{\mathtt{fun.targ}}$ (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. 2.2 is: (1) compute the automaton $n^* = (n|_{\mathtt{fun.targ}}) \sqcap (n|_{\mathtt{arg.type}})$, capturing all *shared* formal and actual parameter types; (2) intersects the root with n^* at *both* paths involved in the constraint: $\left(n|^{\sqcap n^*}_{\mathtt{fun.targ}}\right) |^{\sqcap n^*}_{\mathtt{arg.type}}$. We extrapolate this description into a general algorithm for static reduction:

Definition 2.3.18 (Static Reduction)**.** *Let $c = \{p_1 = \dots = p_k\}$ be a prefix-free PEC; then*

$$\mathsf{reduce}(e, c) = e|^{\sqcap n^*}_{p_1} \dots |^{\sqcap n^*}_{p_k} \qquad \textit{where} \quad n^* = \prod_{p_i \in c} e|_{p_i}$$

Theorem 2.3.19 (Completeness of Reduction)**.** $\mathsf{reduce}(e, c)$ *satisfies the reduction criterion for c.*

Theorem 2.3.20 (Soundness of Reduction)**.** *Let* $e = \mathbf{\Pi}(s, \overline{n}, C)$ *be a transition and* $c \in C$*; then:*

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

2.4 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 Sec. 2.2.4, 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 [6, 73], 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. 2.7, 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. 2.4 (the simplified ECTA encodes all well-typed size-two terms in the environment $\Gamma = \{x\colon \texttt{Int}, y\colon \texttt{Char}, g\colon \alpha \to \alpha, h\colon \texttt{Char} \to \texttt{Bool}\}$).

2.4.1 Enumeration State

The syntax of enumeration states is shown in Fig. 2.8. Var is a countably infinite set of variables, with a dedicated "root" variable $v_\top \in \mathsf{Var}$. 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. 2.7 (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`.

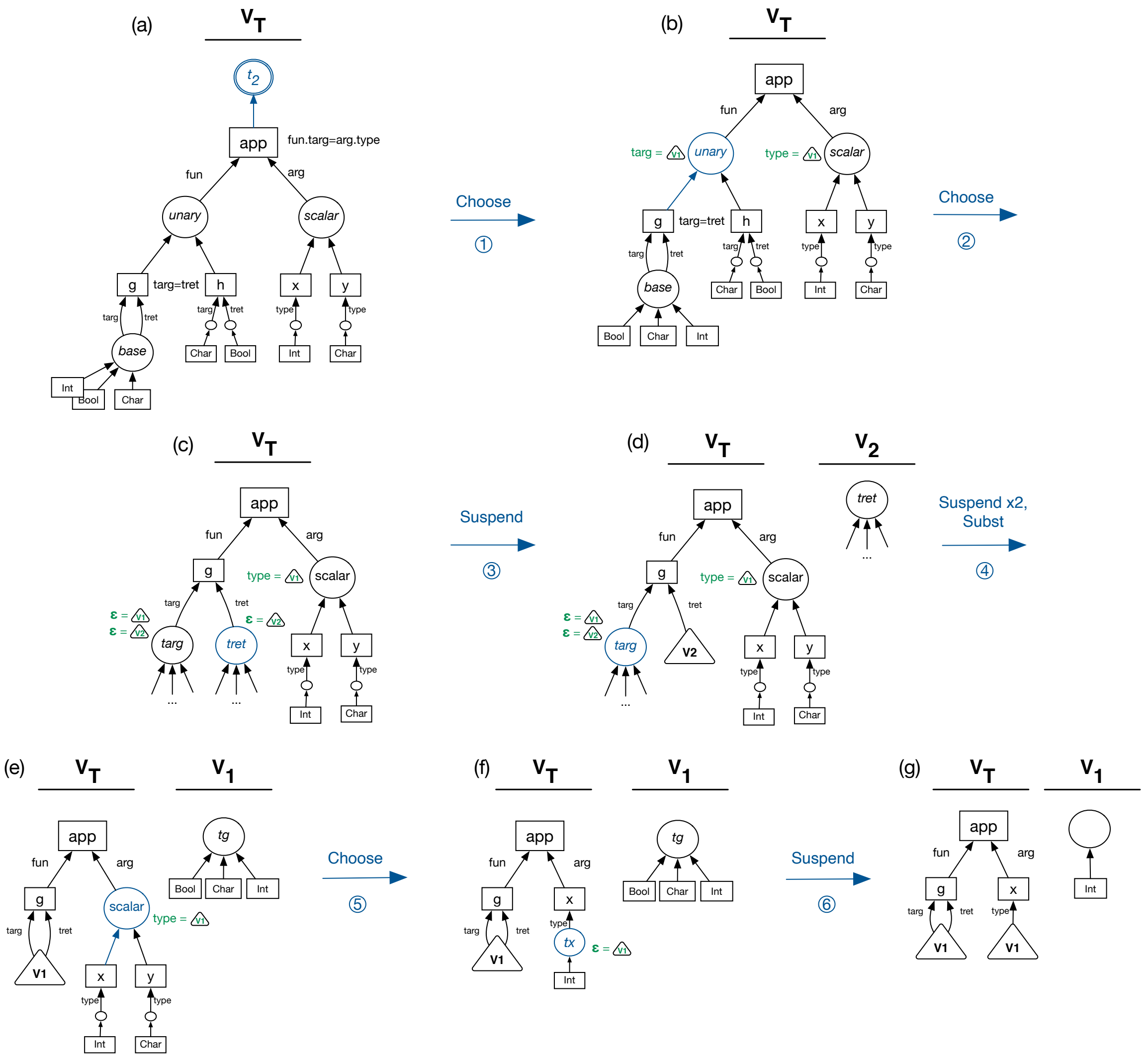


Figure 2.7. An example sequence of steps through enumeration states. The focus node of each step is highlighted in blue and constraint fragments are highlighted in green. The final state is fully enumerated.

Enumeration States

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

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

Figure 2.8. Enumeration states.

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. 2.7 (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 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. 2.7 (b)–(f) and solved in Fig. 2.7 (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 Sec. 2.4.4, this enables compact representation of enumeration results with "trivial differences."

Denotation. The denotation of an enumeration state $[\![\sigma]\!]^S$ is a set of substitutions $\rho : \mathsf{Var} \rightharpoonup \mathcal{T}(\Sigma)$, which is compatible with the constraint fragments and subterm relations imposed by variables

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

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

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

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

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

Figure 2.9. Enumeration rules.

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.

2.4.2 Enumeration Rules

Fig. 2.9 formalizes the above-mentioned CHOOSE and SUSPEND rules as a step relation $\xrightarrow{\sigma} \sigma$ over enumeration states and an auxiliary step relation $\xrightarrow{\tau} \tau'$ 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. 2.7 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 in Fig. 2.10. 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 $\langle \varepsilon = v_1 \rangle$. 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 $\langle \varepsilon = v_1 \rangle$ from *unary*, and also acquires a new fragment $\langle \varepsilon = v_2 \rangle$ by splitting the constraint on `g`.

Finally, consider the rule CHOOSE, which lifts CHOOSE-$\square$ 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. 2.7 (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. 2.7 (g)).

SUSPEND. The SUSPEND rules handle u-nodes with ε*-fragments*, *i.e.* constraint fragments of the form $\langle \varepsilon = v \rangle$.[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 be a u-node, because CHOOSE is not allowed to operate under unsolved variables.

[9] CHOOSE-$\square$ 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.

An example application of SUSPEND-1 is step ③ of Fig. 2.7. The target node *tret* has an ε-fragment $\langle \varepsilon = v_2 \rangle$; since v_2 is uninitialized, SUSPEND-1 creates a new mapping $[v_2 \mapsto tret]$. Step ⑥, on the other hand, is an example of SUSPEND-2: the target node *tx* is restricted by $\langle \varepsilon = v_1 \rangle$; since v_1 already maps to *tg*, SUSPEND-2 updates it with $tx \sqcap tg$. 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. 2.7, which consists of three atomic steps. The first step suspends *targ*, which has *not one but two* ε-fragments—$\langle \varepsilon = v_1 \rangle$ and $\langle \varepsilon = v_2 \rangle$—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 $\langle \varepsilon = v_1 \rangle$. Because of that, SUSPEND-1 can now fire on tg', creating the state $[v_\top \mapsto \ldots, v_2 \mapsto v_1, v_1 \mapsto tg]$, 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. 2.10). After applying SUBST, we arrive at the state in Fig. 2.7 (e).

2.4.3 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

Projecting constraint fragments

$$\begin{aligned}
\mathsf{project}(\Phi, i) &= \textstyle\bigcup_{\phi\in\Phi} \mathsf{project}(\phi, i) \\
\mathsf{project}(\langle c = v\rangle, i) &= \begin{cases} \{\langle c' = v\rangle\} & \text{if } c' \neq \emptyset \\ \emptyset & \text{otherwise} \end{cases} \\
\text{where } c' &= \textstyle\bigcup_{p\in c} \mathsf{project}(p, i) \\
\mathsf{project}(p, i) &= \begin{cases} \bot & \text{if } p = \varepsilon \\ \{p'\} & \text{if } p = i.p' \\ \emptyset & \text{otherwise} \end{cases}
\end{aligned}$$

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

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

Figure 2.10. Auxiliary definitions

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

Theorem 2.4.2 (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:*

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

2.4.4 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. 2.7 does not motivate this decision very well: the fully

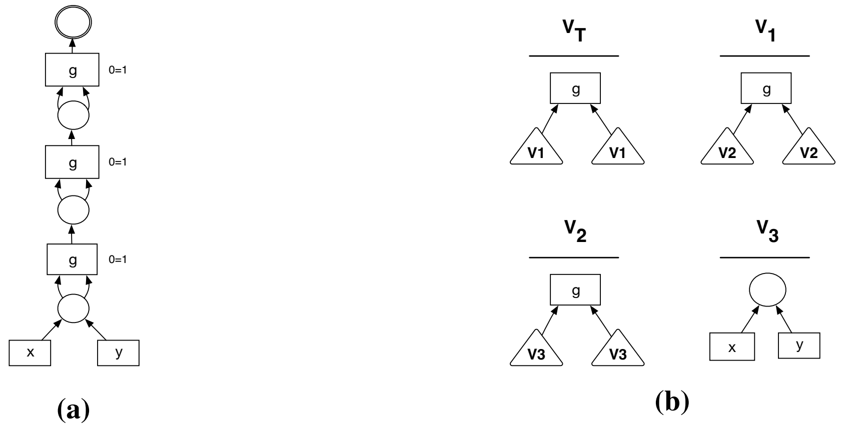


Figure 2.11. (a) An ECTA representing all perfect trees of depth three, whose leaves are either all x or all y (b) The ECTA fully enumerated, in logarithmic space

enumerated state in Fig. 2.7 (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. 2.11a. 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. 2.11b 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 (Sec. 2.7.1), 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

[10]Or, with the cyclic ECTAs of Sec. 2.5, infinitely many.

Syntax

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

Unfolding Recursive Nodes

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

Denotation

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

Figure 2.12. Cyclic ECTAs: syntax and semantics. Here $s \in \Sigma$, C is a PCS, and x is a bound variable.

need not always be fully instantiated, as long as we know that a compatible instantiation exists. In fact, as we explain in Sec. 2.7.2, cyclic ECTAs can encode infinitely many possible polymorphic instantiations, and enumerating them all would be simply impossible.

2.5 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 Sec. 2.7.2) 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.

2.5.1 Cyclic ECTAs: Core Definition

We extend acyclic ECTAs to cyclic by adding "recursive nodes" $\mu x.\mathtt{U}(\overline{e})$. Within this node, x is a variable bound to $\mathtt{U}(\overline{e})$. In diagrams, we depict any use of x as a back-edge to $\mathtt{U}(\overline{e})$ 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 $[\mathtt{U}(\overline{e})/x]n$—or rather, to $[\mu x.\mathtt{U}(\overline{e})/x]n$, since $\mathtt{U}(\overline{e})$ contains further uses of x. Fig. 2.13a shows an cyclic example ECTA, with a recursive node *Nat* representing arbitrary natural numbers defined by the grammar $\mathtt{Nat} ::= S(\mathtt{Nat}) \mid Z$. Fig. 2.12 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

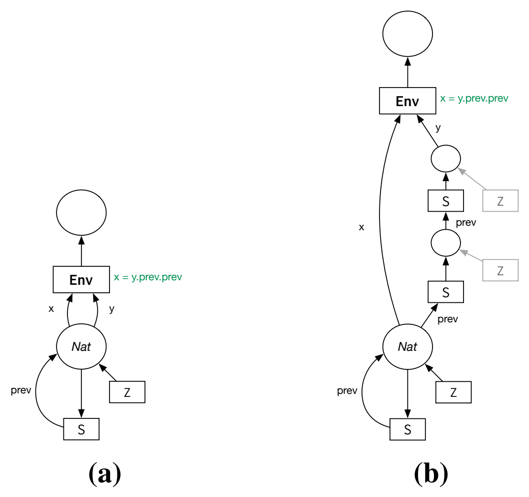


Figure 2.13. (a) ECTA representing an environment with two arbitrary natural numbers x and y, where $y = x + 2$. The `Nat` node is represented $\mu x.\mathtt{U}(\mathbf{\Pi}(S,x),\mathbf{\Pi}(Z))$. (b) The ECTA unfolded into lasso form. The grayed-out transitions will be removed by static reduction.

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]

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

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

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

Definition 2.5.2 (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*:

[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

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

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

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

Figure 2.14. Extensions to prior algorithms to account for recursive nodes

Definition 2.5.3 (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. 2.13b.

2.5.2 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 node pairs, 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 $\mathsf{unfold}(n_1) = \mathtt{U}(\overline{e_1})$ and $\mathsf{unfold}(n_2) = \mathtt{U}(\overline{e_2})$. Then:

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

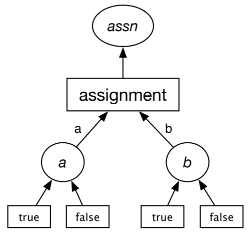
Let $e_1 = \mathbf{\Pi}(s_1, [n_1^0 \dots n_1^{k-1}], C_1)$, $e_2 = \mathbf{\Pi}(s_2, [n_2^0 \dots n_2^{l-1}], C_2)$ be two transitions, then:

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

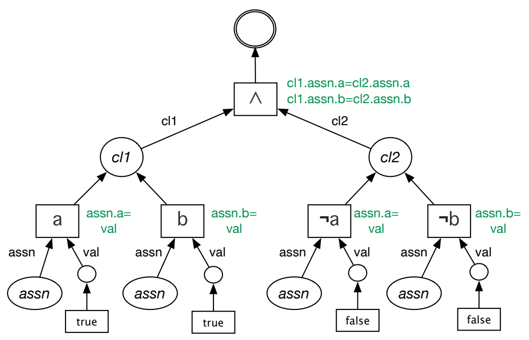
Now, define $n_1 \sqcap n_2 = n_1 \underset{\emptyset}{\sqcap} n_2$.

Static Reduction. Recall from Sec. 2.3.5 that static reduction is defined in terms of intersection at a path, which in turn relies on the definition of nodes at path. Fig. 2.14 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. 2.14 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.



(a) Variable assignment



(b) CNF formula

Figure 2.15. ECTA encoding of a CNF formula $(a \vee b) \wedge (\neg a \vee \neg b)$

2.6 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 carefully optimized, and features heavy memoization based on a mutable hashtable library.

2.7 Applications

This section gives two examples of problem domains that can be reduced to ECTA enumeration: boolean satisfiability (SAT; Sec. 2.7.1) and type-driven program synthesis (Sec. 2.7.2). The second domain has already been introduced informally in Sec. 2.2; here we present its encoding in full generality, and in Sec. 2.8 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.

2.7.1 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. 2.15 illustrates our ECTA encoding for the above formula. The sub-automaton *assn* in Fig. 2.15a 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. 2.15b; 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 Sec. 2.4 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 Sec. 2.4.4).

The overall solving procedure amounts to choosing a literal from each clause and backtracking whenever the assignment becomes inconsistent. For example, suppose ENUMERATE

[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 Sec. 2.10.

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 $\neg$`a` from *cl2*, this inconsistency is discovered immediately when $\neg$`a.val` is suspended and intersected with v_a.

2.7.2 Type-Driven Program Synthesis

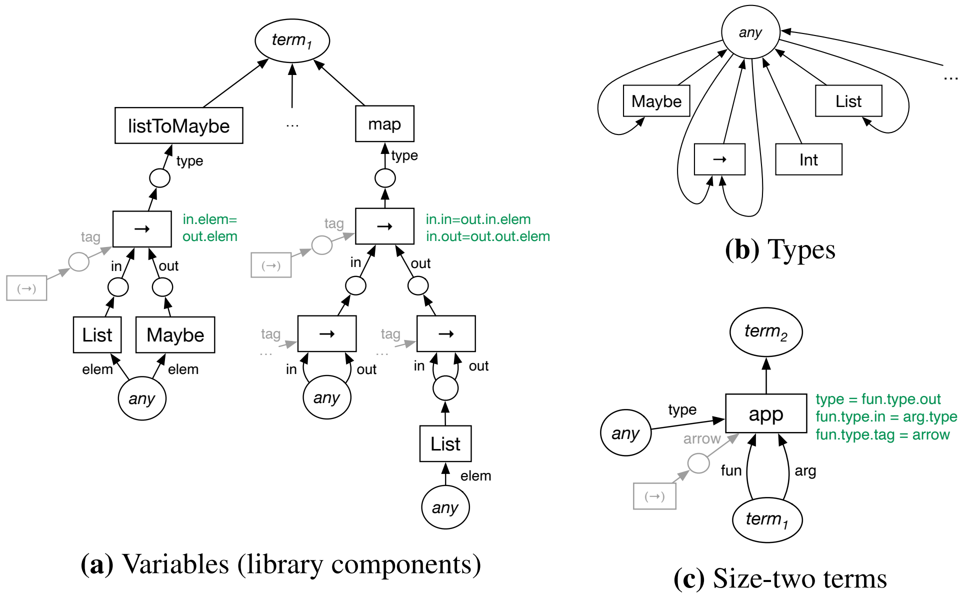
Problem Statement. We are interested in the following *type-driven program synthesis* problem:[13] given a type T, called the query type, and a components library Λ, which maps component names to their types, enumerate terms of type T built out of compositions of components from Λ. For example, a Haskell 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` $\rightarrow$ `[Maybe a]` $\rightarrow$ `a`. Given this problem, the state-of-the-art type-driven synthesizer HOOGLE+ [44, 51] returns a list of candidate programs that includes the desired solution: λ`def mbs` $\rightarrow$ `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: *i.e.* arguments to higher-order functions can be partial applications but not lambdas.

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

[13]This problem is also known as *type inhabitation* [98] and *composition synthesis* [48].



(a) Variables (library components)

(b) Types

(c) Size-two terms

Figure 2.16. Encoding of variables, types, and fixed-size terms in HECTARE.

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 Sec. 2.2 (Fig. 2.4) 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. 2.16b. 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 Sec. 2.2 splits components into different nodes by their arity (*e.g.* the nodes *scalar* and *unary* in Fig. 2.2); this was necessary given our simplified encoding of function types, but as we mentioned above, such arity-specific encoding precludes partial applications. Fig. 2.16a 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 $\to$ transition with two child types, `in` and `out` (for now, ignore the grayed out edges labeled `tag`, we explain those below). Fig. 2.16a showcases the type encoding for two polymorphic components: `listToMaybe` :: $[\alpha] \to$ `Maybe` α and `map` :: $(\alpha \to \beta) \to [\alpha] \to [\beta]$; as before, all occurrences of the same type variable are related by equality constraints, shown in green.

Encoding Applications. Fig. 2.16c 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. 2.16. 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. 2.3. 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 [27, 44] 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.

[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 Sec. 2.2. 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.

In our example, there are four nodes at each term size: $\mathit{term}_n^{\{\mathtt{def},\mathtt{mbs}\}}, \mathit{term}_n^{\{\mathtt{def}\}}, \mathit{term}_n^{\{\mathtt{mbs}\}}, \mathit{term}_n^{\emptyset}$, each representing terms that must mention the corresponding set of inputs. When constructing a new term node, say $\mathit{term}_2^{\{\mathtt{def}\}}$, HECTARE considers all applications of term_1^P to term_1^Q such that $P \cup Q = \{\mathtt{def}\}$. At the top level, only $\mathit{term}_n^{\{\mathtt{def},\mathtt{mbs}\}}$ is connected to the accepting node. Although the 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.

2.8 Evaluation

As we explained in Sec. 2.7.2, 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 [44]. 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.

2.8.1 Comparison on HOOGLE+ Benchmarks

Experiment Setup. For our main experiment, we compare the two synthesizers on the benchmark suite from the latest HOOGLE+ publication [51]. This suite includes 45 synthesis queries, and a

Table 2.1. 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`	`\e → either Left id e`
	`headLast`	`[a] → (a, a)`	`\xs → (head xs, last xs)`
	`both`	`(a → b) → (a, a) → (b, b)`	`\f p → (f (fst p), f (snd p))`
STACK-OVERFLOW	`multiIndex`	`[a] → [Int] → [a]`	`\xs is → map ((!!) xs) is`
	`splitOn`	`Eq a => a → [a] → [[a]]`	`\x xs → groupBy (on (&&) (/= x)) xs`
	`matchedKeys`	`(b → Bool) → [(a, b)] → [a]`	`\p xs → map fst (filter (p . snd) xs)`

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. 2.1. 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 Sec. 2.4.3, repeating this operation up to 30 rounds or until the automaton converges.

Results. Fig. 2.17 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. 2.18, 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+. Fast synthesis times are especially important if a synthesizer is to

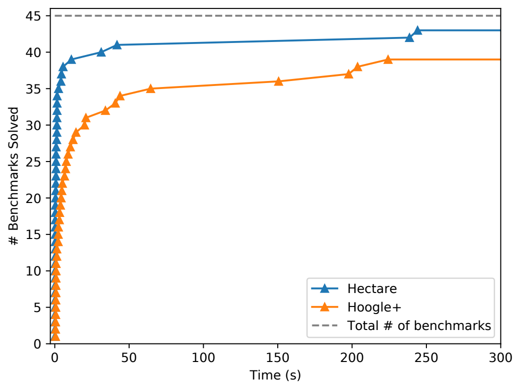


Figure 2.17. Benchmarks solved vs time for HECTARE and HOOGLE+ on HOOGLE+ benchmarks.

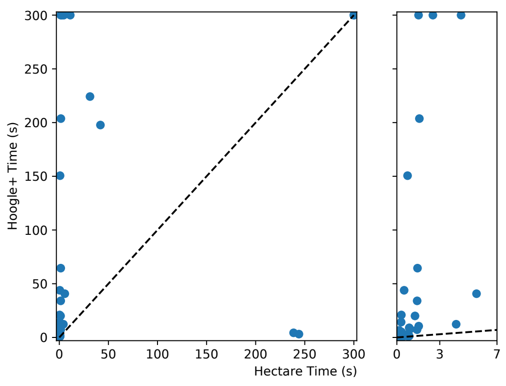


Figure 2.18. Synthesis times of HECTARE against HOOGLE+ on HOOGLE+ benchmarks.

be used interactively. As shown in the zoomed-in scatter plot in Fig. 2.18 (right), HECTARE also vastly outperforms HOOGLE+ if we consider a shorter timeout of 7 seconds, commonly used for interactive synthesizers [28]; in fact, HECTARE solves 84% of the benchmarks within 7 seconds.

The poor performance of HOOGLE+ can be mainly attributed to the brittleness of the abstraction-refinement loop it uses to support polymorphic components (Sec. 2.7.2). For example, the `headLast` benchmark from Tab. 2.1 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. 2.1 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*.

2.8.2 Comparison on StackOverflow Benchmarks

Benchmark Selection. Recall that another limitation of Hoogle+ we discussed in Sec. 2.7.2 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. 2.1.

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. 2.19. Although the search space of HplusAll does include all the new benchmarks, it still fares poorly, solving only

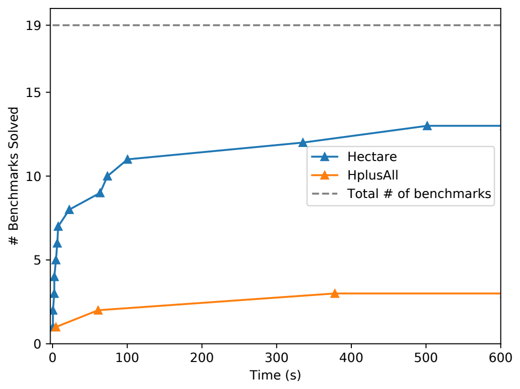


Figure 2.19. Synthesis time comparison on higher-order benchmarks between HECTARE and HPLUSALL.

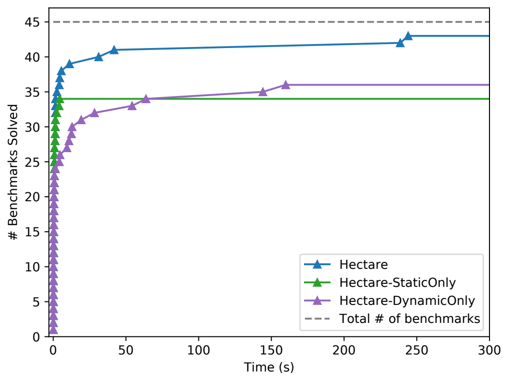


Figure 2.20. Comparison of synthesis performance between HECTARE and its two variants.

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$\times$ 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*.

2.8.3 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. 2.20 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. 2.20 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.*

2.9 Related Work

Constrained Tree Automata. Tree automata have long been used to represent sets of terms in term rewriting [20, 29, 34], 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* [21]. 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 [11, 12, 8, 9, 87] 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 [57, 76, 99] augment context-free grammars with a number of equations of the form $\langle\text{attribute}\rangle = \langle\text{expression}\rangle$. 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 [102, 104, 40, 83, 70, 71, 105] and related areas, such as theorem proving [23], superoptimization [107], and semantic code search [85]. 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 [60]; 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. 2.1, an e-graph equivalent to the FTA in Fig. 2.1b 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.

2.10 Conclusions and Future Work

This chapter 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 work 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 Sec. 2.7.2), 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. 2.11); 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 Sec. 2.7.2, 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 2.5.1 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.

2.11 Acknowledgements

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 work 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.

This chapter, in part, is a reprint of the material as it appears in Proceedings of the ACM on Programming Languages. Volume 6 (ICFP), Article 91. James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, Nadia Polikarpova. ACM 2022. The dissertation author was a principal author and investigator of this paper.

Chapter 3

Type-Directed Progrom Synthesis for REST APIs

3.1 Introduction

Software-as-a-service has emerged as a widely-used means for developers to leverage third-party software. Developers might send requests to STRIPE to handle payments or integrate with SLACK to publish notifications, all while making use of cloud providers to provision various form of storage and compute. According to recent industry surveys, more than 80% of respondents' services offer RESTful APIs [93, 84], and these APIs are extensive. SLACK, for example, has 174 API methods as of version 1.5.0. Amazon Web Services offers over two hundred products and services, each with tens or hundreds of API methods. Even with comprehensive documentation—which is by no means guaranteed—using a new service can be a daunting proposition.

As an example, consider a question posed on STACKOVERFLOW about the SLACK API: *How do I retrieve all member emails from a* SLACK *channel with a given name?* The answer is surprisingly complicated:

1. First, call `conversations_list`[1] to retrieve the array of all channel objects, and then search for a channel object with a given name and get its ID;

[1]We shorten method names for brevity and elide the distinction between REST *methods* and *endpoints*, irrelevant in this context.

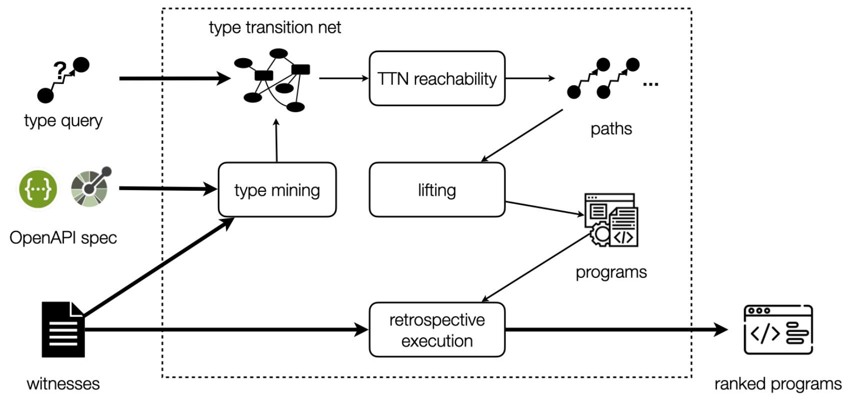


Figure 3.1. Overview of APIPHANY

2. Next, call `conversations_members` on the channel ID to get all user IDs of its members;

3. Finally, for each user ID, call `users_info` to retrieve a user object `u`, and then access the user's email via `u.profile.email`.

To come up with this solution, one must be familiar with `channel` objects, `user` objects, and three different API methods.

Component-based program synthesis [64, 47, 27, 52] has been previously used to help programmers navigate APIs in Java, Scala, and Haskell. Component-based synthesizers take as input a type signature and (in most cases) a set of input-output examples, and return a list of program snippets that compose API calls and have the desired type and input-output behavior. This is a powerful approach for navigating APIs, because it allows developers to start with information easily at hand—the types of inputs they have and the outputs they desire—and requires no knowledge of which API methods to apply.

Challenges. Unfortunately, there are three significant challenges in applying component-based synthesis to RESTful APIs. First, component-based synthesis relies on *types* both for expressing user intent and for efficient search, but types in REST APIs are quite shallow. For example, in the SLACK API specification, both channel names and emails have type `String`, so our example, which transforms a channel name into an array of emails, would have a very imprecise type signature `String` $\rightarrow$ `[String]`.

Second, RESTful APIs commonly transmit *semi-structured data*, *i.e.* arrays of objects, which may themselves contain nested objects and arrays. As a result, using an API is often not as simple as sequencing together a handful of method calls; instead, the calls must be interleaved with "data wrangling" operations such as projections, maps, and filters. These data wrangling operations are challenging for component-based synthesis: they are extremely generic, and hence significantly expand the search space.

Finally, to compensate for the inherent ambiguity of types, component-based synthesis typically relies on *executing* candidate program snippets and matching them against user-provided input-output examples. In a software-as-a-service environment, this is a complete non-starter: not only is the user generally unaware of the internal state of the service and hence unable to provide accurate examples, but executing API calls during synthesis can also be prohibitively expensive due to rate limits imposed by the services and, even more importantly, can have unrecoverable side effects, such as deleting accounts or publishing messages.

APIPHANY*: synthesis with semantic types.* Our core insight is that type-based specifications are actually a good fit for REST APIs, as long as the types are more fine-grained. In our example, if the SLACK API had dedicated types for `Channel.name` and `Profile.email`, the programmer could specify their intent as the type `Channel.name` $\rightarrow$ `[Profile.email]`. Although this specification is still somewhat ambiguous, intuitively it has enough information to narrow down the synthesis results to a manageable number such that the programmer can manually inspect the remaining solutions. We refer to such fine-grained types as *semantic types*.

In this work, we present APIPHANY, a component-based synthesizer for REST APIs guided by semantic types. Fig. 3.1 shows a high-level overview of our approach, which is structured into two phases: (1) the *analysis* phase infers semantic type annotations for a given API; (2) the *synthesis* phase uses these type annotations to perform component-based synthesis. For the SLACK API, APIPHANY is able to infer, for example, that the method `conversations_members` has the semantic type `Channel.id` $\rightarrow$ `[User.id]`. At synthesis time, given the *type query* `Channel.name` $\rightarrow$

```
\channel_name → {
  c   ← conversations_list()
  if c.name = channel_name
  uid ← conversations_members(channel=c.id)
  let u = users_info(user=uid)
  return u.profile.email
}
```

Figure 3.2. Solution for retrieving all member emails from a SLACK channel in APIPHANY DSL.

`[Profile.email]`, APIPHANY returns a ranked list of programs of this type, where the desired solution (shown in Fig. 3.2) appears among the top ten. APIPHANY's output is expressed in a compact DSL inspired by Haskell's monadic `do`-notation and Scala's `for`-comprehensions, which, however, can be easily translated into the user's language of choice for communicating with the API.

Contributions. We present the design, implementation, and evaluation of APIPHANY, including:

1. *Type mining* (Sec. 3.4), a technique that infers semantic types from a set of *witnesses* (observed invocations of API methods). Witnesses can be generated in a sandbox or by tapping live production traffic; in either case, they are collected ahead of time, once per API, which avoids inducing side effects during synthesis.

2. Efficient synthesis of wrangling operations for semi-structured data via *array-oblivious search* (Sec. 3.5), which omits challenging array operations during search, and recovers them later via type-directed lifting.

3. Ranking synthesis results with the help of *retrospective execution* (Sec. 3.6), a type of simulated execution using previously collected witnesses. Retrospective execution helps APIPHANY weed out uninteresting programs (*e.g.* programs that always return an empty array), reducing the number of synthesis results the user has to inspect to find their expected solution.

```
{"user": {"type": "object",
          "properties": {"id":{"type": "string"},
                         "name": {"type": "string"},
                         "profile":{"$ref": "#/definitions/profile"}}},
 "profile": {"type": "object",
             "properties": {"display_name": {"type": "string"},
                            "email": {"type": "string"}}},
 "channel": {"type": "object",
             "properties": {"creator": {"type": "string"},
                            "name": {"type": "string"},
                            "id": {"type": "string"}}}
}
```

```
{"users_info":
  {"parameter": [
    {"in": "query", "name": "user", "type": "string"}],
   "responses": {"200": {"schema": {"properties":
      {"user": {"$ref": "#/definitions/user"}}
  }}}},
 "conversations_list":
  {"parameter": [],
   "responses": {"200": {"schema": {"properties":
     {"channels":
      {"type": "array",
       "items": {"$ref": "#/definitions/channel"}
      }}
  }}}}
}
```

Figure 3.3. Fragment of the Slack API's OpenAPI specification. (left) Definitions of `user`, `profile` and `channel` objects. (right) Parameters and responses of the methods `users_info` and `conversations_list`.

We evaluate APIPHANY on three real-world APIs, and 32 tasks extracted from GITHUB repositories and STACKOVERFLOW (Sec. 3.7). Our evaluation shows that APIPHANY can find solutions to the majority of tasks (29/32) within 150 seconds. Moreover, semantic types are crucial to its effectiveness: without type mining, APIPHANY can only solve four tasks. Finally, ranking significantly improves the quality of reported solutions, increasing the number of correct solutions appearing in top ten results from 12/29 to 23/29.

3.2 APIPHANY by Example

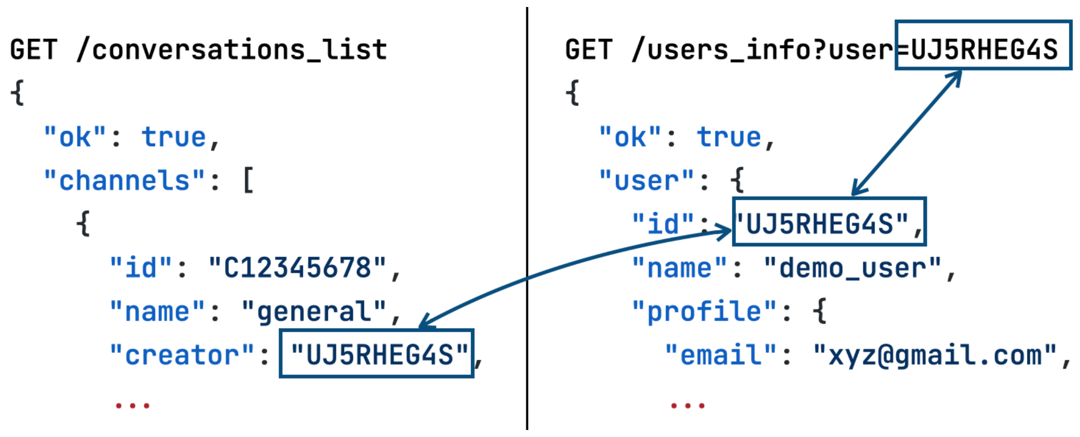
In this section we use the task of retrieving all member emails in a SLACK channel as a running example to illustrate the APIPHANY workflow depicted in Fig. 3.1.

3.2.1 API Analysis by Example

API analysis is performed once per API. It takes as input a *spec* in the popular OpenAPI format[2] and a set of *witnesses* (successful API method calls); it produces a spec annotated with semantic types. OpenAPI specs are publicly available for most popular APIs.[3] Witnesses can be generated in a number of ways, for example, by running an integration test suite in a sandbox or by passively listening to production API traffic. We envision witness collection and API analysis

[2]https://swagger.io/. APIPHANY supports both OpenAPI v2 and v3.

[3]SLACK OpenAPI spec is available at: https://raw.githubusercontent.com/slackapi/slack-api-specs/master/web-api/slack_web_openapi_v2.json



```
GET /conversations_list
{
  "ok": true,
  "channels": [
    {
      "id": "C12345678",
      "name": "general",
      "creator": "UJ5RHEG4S",
      ...
```

```
GET /users_info?user=UJ5RHEG4S
{
  "ok": true,
  "user": {
    "id": "UJ5RHEG4S",
    "name": "demo_user",
    "profile": {
      "email": "xyz@gmail.com",
      ...
```

Figure 3.4. Witnesses for two SLACK API methods. Arrows connect equal values observed at different locations. Type mining ascribes the type `User.id` to all the boxed locations.

being performed by the API maintainer (or another interested party), not by regular users of the APIPHANY synthesizer.

OpenAPI specs. Fig. 3.3 shows a fragment of the OpenAPI spec provided by SLACK. An OpenAPI spec consists of object definitions and method definitions. We show definitions of three objects, `user`, `profile` and `channel`, and two methods, `users_info` and `conversations_list`, relevant to our example. As you can see, the spec does provide precise type information for some of the locations: for example, the response of `users_info` clearly has type `User` (it is annotated with a reference to the corresponding object definition). The bulk of the locations, however, such as the field `user.id` or the parameter of `users_info`, are simply annotated with `String`, which is not very helpful for the purposes of type-directed synthesis. Our goal is to replace these `String` annotations with more fine-grained types.

Mining types from witnesses. To this end, we build upon an algorithm first proposed in [3] that infers types by *mining* them from execution traces, based on the insight that *equal values observed at different locations likely have the same type*. More specifically, our type mining algorithm starts by ascribing a unique semantic type to each `String` location and then merges locations that share a value anywhere in the witness set. As an illustration, consider Fig. 3.4, which lists two witnesses for the API methods from our running example. In this witness set we observe the same value `"UJ5RHEG4S"` in three locations: (1) the *parameter* of `users_info`, (2) the `id` *field* of a `User` object (we know from the spec that `users_info` returns a `User`), and (3) the `creator` *field* of a `Channel` object (we know from the spec that `conversations_list` returns an array of `Channels`). Hence we merge all three locations into the same semantic type. For presentation purposes, we assign the name `User.id` to this type, which is derived from location (2) above. The choice of name is not important, however: the user is free to refer to this semantic type via any of its representative locations; for example, `Channel.creator` also denotes the same type.

3.2.2 Program Synthesis by Example

The program synthesis phase of APIPHANY is meant to be used by regular programmers, any time they need help accomplishing a task with one of the supported APIs. The programmer queries APIPHANY with a type signature built from semantic types. Although the UI for constructing queries is beyond the scope of this work, we envision the programmer browsing object definitions and selecting relevant fields as semantic types. For our running example, the programmer knows that they need to go from a channel name to an array of user emails; they might first look through the `channel` object definition and find the `name` field; they might then search globally for a field called `email` and find it inside the `profile` object; hence they settle on the type query `Channel.name` $\rightarrow$ [`Profile.email`].

The program synthesis phase itself comprises two steps, beginning with a program *search* step to generate a list of candidate programs with a given type, followed by a *ranking* step to identify promising candidates (described in Sec. 3.2.3).

Challenge: components meet control flow. Given the type query `Channel.name` $\rightarrow$ [`Profile.email`], how would APIPHANY go about enumerating all programs of this type? This task presents a challenge to existing synthesis techniques because our candidate programs have *both* a large component library to choose from—from dozens to hundreds of methods—*and* non-trivial control flow—*e.g.* the solution to our running example has to *loop* over the members of a channel. One line of prior work that scales to large component libraries is graph-based search using *type-transition nets* (TTNs) [27, 45]; unfortunately, this approach can only generate sequences of method calls, and does not support loops.

The* APIPHANY *DSL. We observe that the loops we need for manipulating semi-structured data are restricted to iterating over (possibly nested) arrays of objects. To capture this restricted class of programs we have designed a DSL inspired by Scala's `for`-comprehensions, Haskell's monadic `do`-notation, and `LINQ` [65]. The solution to our running example in this DSL is given in Fig. 3.2. In this language, iteration over an array is expressed using the monadic *bind* operation (written ←). For example, the second bind in Fig. 3.2 has the effect of performing the subsequent computation for every element `uid` of the array returned in line 4:

```
4  uid  ← conversations_members(channel=c.id);
5  let u = users_info(user=uid);
6  return u.profile.email
```

Array-oblivious search. The main idea behind APIPHANY's search is that although we cannot directly synthesize the program above using existing TTN-based techniques, we can synthesize an *array-oblivious* version of this program, where we pretend that `conversations_members` returns a single `User.id` instead of an array, and hence we can simply sequence the two method calls, without monadic binding:

```
4  let uid = conversations_members(channel=c.id);
5  let u   = users_info(user=uid);
6  u.profile.email
```

To transform an array-oblivious program into the final solution, APIPHANY *lifts* it into a comprehension by replacing each **`let`** binding that causes a type mismatch with a monadic bind. In our example, the **`let`** in line 4 causes a type error (because `conversations_members` returns `[User.id]`, while `users_info` expects a single `User.id`), while the **`let`** in line 5 does not (since `users_info` returns a single `User`); hence lifting replaces the first **`let`**-binding with ← but not the second.

```
\channel_name → {
  c ← conversations_list()
  if c.name = channel_name
  let uid = c.creator
  let u = users_info(user=uid)
  return u.profile.email
}
```

```
\channel_name → {
  c ← conversations_open()
  if c.name = channel_name
  let uid = c.creator
  let u = users_info(user=uid)
  return u.profile.email
}
```

Figure 3.5. A sample of incorrect candidate solutions.

3.2.3 Ranking via Retrospective Execution

Although semantic types are less ambiguous than primitive types for expressing user intent, they are still not precise enough to exactly identify the desired program. For example, our synthesizer generates more than 1000 candidates for the type signature `Channel.name` → `[Profile.email]`; clearly, it is infeasible for the user to manually go through all of them. Hence, APIPHANY must be able to rank the candidates in order to show the user a small number of likely solutions.

Fortunately, most of the 1000 candidates are easy to weed out because they produce uninteresting results. Consider two of the candidates depicted in Fig. 3.5, which differ from our desired solution (Fig. 3.2) in the highlighted fragments: the first program returns the email of the channel's *creator* (as opposed to all of its members), and the second one gets the list of channels from `conversations_open`, which is intended for opening a direct message channel. It turns out that the second program *always fails* at run time, because a successful call to `conversations_open` requires providing exactly one of its two optional arguments (a channel ID or a list of users). The first program executes successfully, but it always returns a single email, while the user asked for an array of emails. For these reasons, both of these programs are less likely to be the intended solution than the program in Fig. 3.2, which successfully returns multiple emails at least sometimes.

A natural idea is to test all candidate programs on random inputs and rank them based on the results they produce. Unfortunately, as we have hinted above, there are several barriers to systematically executing many candidate programs that make calls to REST APIs. First, most REST APIs set a rate limit on how frequently a user can make method calls or how many calls a user can make in a day. Second, many REST API methods are side-effecting. Unlike a self-contained binary, a remotely-hosted service cannot be restarted from a clean state for each execution.

Retrospective execution. We propose *retrospective execution* (RE) as an efficient, non-side effecting alternative to program execution. The main idea is to simulate execution by "replaying" witnesses collected for the API analysis phase. When evaluating a candidate program, rather than executing an API call, RE instead searches for a matching witness and substitutes its response at the call site. If done naively, however, this process almost always yields failure or an empty array; so making RE useful for ranking purposes requires explicitly *biasing* execution towards meaningful results.

As an illustration, consider executing the program in Fig. 3.2 using the witnesses in Fig. 3.4. As the first step, we simulate the call to `conversations_list` using the first witness; the response is an array of channels with names `"general"`, `"private-test"`, and `"team"`. The second step is to filter this array, retaining only those channels whose name is equal to the input parameter `channel_name`. If we had sampled the value for `channel_name` eagerly, before running the program, we could scarcely have chosen one of the three names actually present in the array, so the filtering step (and hence the whole program) would almost always return an empty array. Instead we sample the value for `channel_name` *lazily*, once we encounter the filter, picking one of the names present in the array.

Assume that we picked `channel_name = "general"`, and hence the filter returns the first channel. Next, we simulate the call to `conversations_members` on this channel's ID. Because our witness set is sparse, we may or may not find an exact match for this call; in the latter case, we sample the response from the set of *approximate matches*, *i.e.* witnesses with the same method names and argument names,[4] but not necessarily the same argument values. Due to approximate

[4]Because in REST some arguments are optional, the same method can be called with different subsets of arguments.

matching, RE results do not always equal the results of a real execution, but they are still useful for estimating whether a program candidate is able to produce meaningful outputs. For each candidate, we run RE multiple times (with different random seeds) and use the outputs to assign a rank to each candidate.

3.3 The Core Language

In this section, we formalize the core of APIPHANY's DSL as λ_A, a functional language specialized for manipulating semi-structured data. The syntax of λ_A is summarized in Fig. 3.6.

Types. The types of λ_A include *syntactic types* t (those used in the OpenAPI spec) and *semantic types* $\hat{t}$, which we infer. Both categories of types have named objects o, arrays $[t]$, and records $\{\overline{\ell_i : t_i}\}$.[5] Records are mappings from field labels to types; some fields are optional, indicated with a ? before its label. For example, the record type $\{\texttt{id} : \texttt{String}, ?\texttt{time\_zone} : \texttt{String}\}$, has a required field `id` and an optional field `time_zone`. The two categories of types differ in their base types: the sole primitive syntactic type is `String`,[6] while the sole primitive semantic type is a *loc-set*, *i.e.* a set of locations.

A *location* is an object or method name followed by a sequence of labels, such as `User.id`. Apart from field labels that correspond to object fields in the OpenAPI spec, we introduce three reserved labels—in, out, and 0—for addressing method parameters and responses, and array elements, respectively. For example, `c_list`.out.0 refers to an element type of the response array of the method `c_list`.

Function types are written $t \to t$, and multiple arguments are represented as a record whose fields encode argument names (with optional fields encoding optional arguments).

A library Λ models an OpenAPI spec. It contains object definitions, which bind object identifiers to (record) types, and method definitions, which bind method names to function

[5]We write $\overline{X}$ to denote zero or more occurrences of a syntactic element X.

[6]In practice, REST APIs also include integers and booleans; these types are handled slightly differently in APIPHANY, as discussed in Sec. 3.7.4.

$o ::=$	`User` $\mid$ `Channel` $\mid \ldots$	object names
$f ::=$	`u_info` $\mid \ldots$	method names
$l ::=$	in $\mid$ out $\mid$ 0 $\mid$ `id` $\mid$ `name` $\mid \ldots$	field labels
$\ell ::=$	$l \mid ?l$	record fields
$loc ::=$	$o.\overline{l} \mid f.\overline{l}$	locations

Terms

$e ::=$		*Expressions*
	$\mid x \mid e.l$	variable, projection
	$\mid f(\overline{l_i = e_i}) \mid$ **`let`** $x = e; e$	method call, pure binding
	$\mid$ **`if`** $e = e; e \mid x \leftarrow e; e$	guard, monadic binding
	$\mid$ **`return`** e	pure value lifting
$\mathcal{E} ::=$	$\lambda \overline{x}.e$	*Top Level Programs*

Values

$v ::=$ `"..."` $\mid [\overline{v}] \mid \{\overline{l_i = v_i}\}$	strings, arrays, objects

Types

$t ::=$		*Syntactic types*
	$\mid$ `String`	strings
	$\mid o \mid [t] \mid \{\overline{\ell_i : t_i}\}$	named objects, arrays, records
$s ::=$	$t \rightarrow t$	function types
$\hat{t} ::=$		*Semantic types*
	$\mid \{\overline{loc}\}$	loc-sets
	$\mid o \mid [\hat{t}] \mid \{\overline{\ell_i : \hat{t}_i}\}$	named objects, arrays, records
$\hat{s} ::=$	$\hat{t} \rightarrow \hat{t}$	function types

Libraries

$\Lambda ::=$	$\overline{o : t}; \overline{f : s}$	object and method definitions
$\hat{\Lambda} ::=$	$\overline{o : \hat{t}}; \overline{f : \hat{s}}$	semantic definitions

Figure 3.6. Syntax of the language λ_A

	Syntactic library Λ	Semantic library $\hat{\Lambda}$

Objects	`Channel: { id: String,`
	`           name: String,`
	`           creator: String }`
	`User: { id: String,`
	`        name: String,`
	`        profile: Profile }`
		`Channel: { id: Channel.id ,`
		`           name: Channel.name ,`
		`           creator: User.id  }`
		`User: { id: User.id ,`
		`        name: User.name ,`
		`        profile: Profile }`

Methods	`c_list:`
	`  {} → [Channel]`
	`u_info:`
	`  {user: String} → User`
	`c_members:`
	`  {channel: String} → [String]`
		`c_list:`
		`  {} → [Channel]`
		`u_info:`
		`  {user: User.id } → User`
		`c_members:`
		`  {channel: Channel.id } → [ User.id ]`

Figure 3.7. Library Λ that models a portion of the SLACK OpenAPI spec and the corresponding semantic library $\hat{\Lambda}$. Each gray box is a loc-set type inferred by type mining, depicted for brevity using a single representative location from the set.

```
Input: A library $\Lambda$ and witnesses $\mathcal{W}$
Output: A semantic library $\hat{\Lambda}$
1: function MINETYPES($\Lambda$, $\mathcal{W}$)
2:   $DS \leftarrow$ empty disjoint-set
3:   for $\langle f, v_{in}, v_{out} \rangle \in \mathcal{W}$ do
4:     ADDWITNESS($DS, f$, in, $v_{in}$)
5:     ADDWITNESS($DS, f$, out, $v_{out}$)
6:   $\hat{\Lambda} \leftarrow$ ADDDEFINITIONS($\Lambda, DS$)
7:   return $\hat{\Lambda}$
```

```
1: function ADDWITNESS($DS, loc, v$)
2:   match $v$
3:     case "...":
4:       $\Lambda; loc \vdash \{loc'\} \Longrightarrow$
5:       $DS \leftarrow$ insert($DS, loc', v$)
6:     case $[\overline{v_i}]$:
7:       forall $i$ : ADDWITNESS($DS, loc.0, v_i$)
8:     case $\{\overline{l_i = v_i}\}$:
9:       forall $i$ : ADDWITNESS($DS, loc.l_i, v_i$)
```

Figure 3.8. Type mining algorithm.

types. A semantic library $\hat{\Lambda}$, which is the output of type mining, binds object identifiers and method names to semantic types. As an example, Fig. 3.7 shows Λ definitions that correspond to a portion of the SLACK OpenAPI spec (with method names shortened for brevity), and their corresponding definitions in the semantic library $\hat{\Lambda}$.

Terms. Values of λ_A include string literals, arrays, and objects; objects are mappings from field labels to values. Similarly to Haskell's `do`-notation, **`return`** e returns an array with a single element e, and the monadic binding $x \leftarrow e_1; e_2$ evaluates e_2 for each element x of the array e_1, and concatenates all resulting arrays. In contrast, the pure binding **`let`** $x = e_1; e_2$ binds x to the entire result of e_1 and then evaluates e_2. The guard expression **`if`** $e_1 = e_2; e$ evaluates e if the guard holds and returns an empty array otherwise; guards are restricted to equalities, since these are the only guards generated by APIPHANY. At the top level, a program $\mathcal{E}$ is an abstraction with a list of arguments $\bar{x}$ and body e.

3.4 Type Mining

In this section we detail APIPHANY's type mining algorithm, using the library Λ in Fig. 3.7 and the witnesses in Fig. 3.4 as a running example. Informally, the idea is to first assign every `String` location loc in Λ a unique type $\{loc\}$, and then merge the types of some locations based on the witnesses.

Assigning location-based types. We formalize the first step as a judgement $\Lambda; loc \vdash \hat{t} \Longrightarrow$, which assigns a semantic type $\hat{t}$ to location loc based only on the information present in the syntactic library Λ. The reader might be wondering why isn't the assigned type $\hat{t}$ always simply $\{loc\}$. This is indeed the case for `String`-annotated locations explicitly present in Λ, such as `User.id` or `u_info`.in.`user`. But in other cases, location-based type assignment is more involved; for example:

- $\Lambda;$ `u_info`.out $\vdash$ `User` $\Longrightarrow$ because this location is annotated with a named object type.
- $\Lambda;$ `c_members`.out $\vdash [\{$`c_members`.out.0$\}] \Longrightarrow$ because array types do not themselves get replaced with loc-sets; instead, we recursively assign a location-based type to an array's element.
- $\Lambda;$ `u_info`.out.`id` $\vdash \{$`User.id`$\} \Longrightarrow$ because type assignment *canonicalizes* locations inside types to make sure they explicitly appear in Λ; to this end, we recursively assign a type to location's prefix, $\Lambda;$ `u_info`.out $\vdash$ `User` $\Longrightarrow$, and then follow the field `id` of the `User` object.

The formalization of location-based type assignment is mostly straightforward and relegated to the technical report [41].

Merging types via a disjoint-set. Type mining relies on a variant of the *disjoint-set* data structure (also known as *union-find* [96]). Our disjoint-set DS stores disjoint groups of pairs (loc, v), where *loc* is a location and v is a string value. When two pairs are in the same group, their corresponding locations have the same semantic type.

DS supports two efficient operations: insert and find. insert takes a pair (loc, v) and checks whether either of its components already appears in *DS*; if so, it merges the new pair into the corresponding group, and otherwise puts it into a new group. find takes a location *loc* and returns a semantic type $\hat{t}$; internally, find locates the group to which the pair $(loc, \_)$ belongs in *DS* and returns the loc-set $\{loc, loc_1, \ldots\}$ that contains all locations in that group.

Type mining algorithm. Fig. 3.8 presents the top-level algorithm MINETYPES, which takes as input a syntactic library Λ and a set of witnesses $\mathcal{W}$, and returns a semantic library $\hat{\Lambda}$. A *witness* W is a triple $\langle f, v_{in}, v_{out} \rangle$, where f is a method name and v_{in}, v_{out} are its argument and response value (multiple arguments are represented as an object). MINETYPES operates in two phases: in lines 2–5 it builds the disjoint-set *DS* from $\mathcal{W}$ and in line 6 it build $\hat{\Lambda}$ from *DS*.

In the first phase, the algorithm iterates over the witnesses, registering the input value v_{in} at the location f.in and the output value v_{out} at the location f.out. To this end, we call a helper function ADDWITNESS, which drills down into composite values (arrays and objects) to get to string literals, and then inserts each string into *DS* with its location-based type. For example, when processing the response from the first witness in Fig. 3.4, ADDWITNESS iterates over all channel objects in the array, and over all fields of each channel object; once it reaches the value `"UJ5RHEG4S"`, it computes the type of its location as $\Lambda; \texttt{c\_list}.\text{out}.0.\texttt{creator} \vdash \{\texttt{Channel.creator}\} \Longrightarrow$, and inserts $(\texttt{Channel.creator}, \texttt{"UJ..."})$ into *DS*. Processing the second witness results in inserting the pairs $(\texttt{u\_info}.\text{in}.\texttt{user}, \texttt{"UJ..."})$ and $(\texttt{User.id}, \texttt{"UJ..."})$, which share the same string value, and hence all three pairs get merged into the same group. Once all the witnesses are added to *DS*, its groups represent the final set of semantic types.

In the second phase, the algorithm calls ADDDEFINITIONS to iterate over all object and method definitions in Λ, and add corresponding definitions to $\hat{\Lambda}$, relying on find to retrieve the semantic type for each location. For example, when adding the method `u_info`, we query $\mathsf{find}(DS, \texttt{u\_info}.\mathsf{in}.\texttt{user})$, which finds the group mentioned above and returns its loc-set: $\{\texttt{User.id}, \texttt{Channel.creator}, \ldots\}$. If the requested location is not in DS—because $\mathcal{W}$ has no witnesses for the enclosing method or object—it is annotated with the unmerged location-based type.

3.5 Type-Directed Synthesis

In this section, we discuss how APIPHANY generates a set of well-typed programs given a query type, using the same running example as in previous sections.

Synthesis problem. Formally, our synthesis problem is defined by a semantic library $\hat{\Lambda}$ and a semantic query type $\hat{s}$. For our running example, we use the semantic library from Fig. 3.7 and the query type $\texttt{Channel.name} \to [\texttt{Profile.email}]$.[7] A *candidate solution* is any program $\mathcal{E}$ that type-checks against $\hat{s}$. To formalize this notion, we introduce the program typing judgment $\hat{\Lambda} \vdash \mathcal{E} :: \hat{s}$, which is mostly straightforward. We note only that in a monadic binding $x \leftarrow e_1; e_2$, both e_1 and e_2 must have array types; in a guard $\textbf{if}\ e_1 = e_2; e$, e must have an array type, while e_1 and e_2 must have (the same) loc-set type, since equality is only supported over string values. Full definition can be found in the technical report [41].

Type transition nets. To efficiently enumerate well-typed programs we follow prior work [27, 45] and encode the search space as a special kind of Petri net, called *type-transition net* (TTN). Intuitively, a TTN encodes how each API method transforms values of one semantic type into another; *e.g.* `u_info` transforms a `User.id` into a `User`. Fig. 3.9 shows a TTN for our running example. Places (circles) correspond to semantic types, transitions (rectangles) correspond to methods, and edges connect methods with their input and output types. In addition to API methods, the TTN contains transitions that correspond to λ_A projections (*e.g.* $\mathsf{proj}_{\texttt{User.profile}}$ and $\mathsf{proj}_{\texttt{Profile.email}}$) and guards (*e.g.* $\mathsf{filter}_{\texttt{Channel.name}}$).

Array-oblivious search. For our search space encoding to be useful, we need to make sure that

[7] Here and throughout this section, we write loc-set types using an arbitrarily chosen representative; the user can query APIPHANY using any locations of their choosing, and the tool interprets them as the loc-sets they belong to.

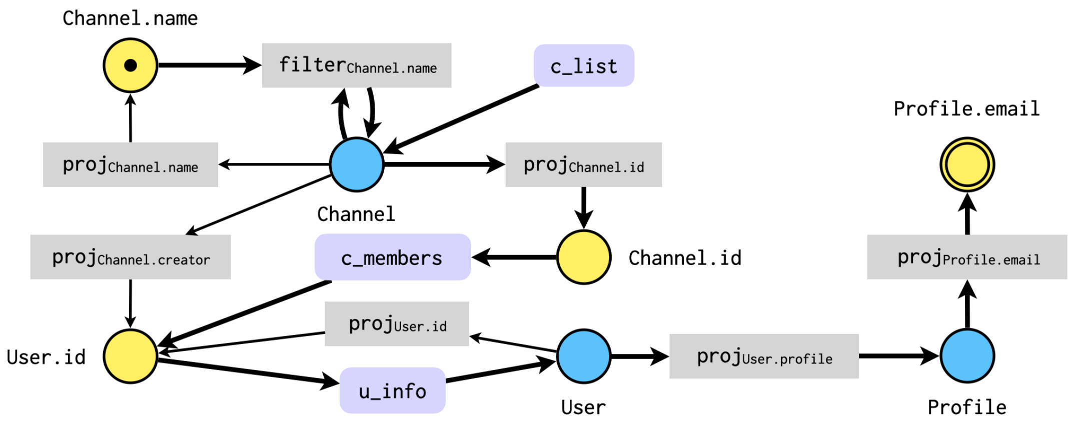


Figure 3.9. A fragment of the type-transition net (TTN) for SLACK. Places (circles) are semantic types; transitions (rectangles) are API methods and data transformations. The bold path represents the solution to our running example.

every well-typed λ_A program corresponds to a path in the TTN. This is where we encounter a challenge: there is no straightforward way to encode λ_A's monadic bind operation into the TTN. Although prior work on HOOGLE+ [45] supports higher-order functions, the arguments to those functions are syntactically restricted to variables (*i.e.* inner lambda abstractions are not supported), which is insufficient for our purposes. To address this problem, we introduce a new, *array-oblivious* TTN encoding, which does not distinguish between array types and types of their elements, and hence does not require monadic binds. For example, in Fig. 3.9 `c_members` returns `User` instead of `[User]`, and hence its output can be passed directly to `u_info`, without iterating over it.

Search in the TTN. Once the TTN is built, we enumerate paths from the input to the output type (or rather, array-oblivious versions thereof). In our example, we place a *token* in the input type `Channel.name` and search for a path (a sequence of transitions) that would get this token to the output type `Profile.email`, possibly generating and consuming extra tokens along the way. The bold path in Fig. 3.9 corresponds to our desired solution from Fig. 3.2. On this path, we first fire the transition `c_list` (which does not consume any tokens) to produce an extra token in `Channel`. Next, we fire filter$_{\texttt{Channel.name}}$, which consumes the two tokens in `Channel` and `Channel.name`, and

Input: Semantic library $\hat{\Lambda}$ query type $\hat{s}$
Output: Set of candidate solutions $\overline{\mathcal{E}}$

```
1: function SYNTHESIZE($\hat{\Lambda}, \hat{s}$)
2:   $\mathcal{N} \leftarrow$ BUILDTTN($\hat{\Lambda}$)
3:   $I, F \leftarrow$ PLACETOKENS($\hat{s}$)
4:   for $\pi \in$ PATHS($\mathcal{N}, I, F$) do
5:     for $\mathcal{E} \in$ PROGS($\pi$) do
6:       yield LIFT($\hat{\Lambda}, \hat{s}, \mathcal{E}$)
```

Figure 3.10. Synthesis algorithm

produces a single token in `Channel`. The remaining five transitions on the bold path simply move this one token along until it reaches `Profile.email`.

Like in prior work [27, 45], a path is only considered valid if the final state contains exactly one token in the output type (and no tokens in any other types); this condition ensures that the generated programs use all their inputs.

Synthesis algorithm. APIPHANY's top-level synthesis algorithm is depicted in Fig. 3.10. The algorithm first constructs a TTN $\mathcal{N}$ and encodes the query type $\hat{s}$ as an initial and final token placement, I and F; it then enumerates all paths from I to F in $\mathcal{N}$ in the order of length (until timeout). For each path π, the algorithm iterates over the corresponding array-oblivious programs $\mathcal{E}$ and *lifts* them into well-typed λ_A programs. The reason π might yield multiple programs is that the TTN does not distinguish different arguments of the same type, and hence we must try all their combinations.

Because TTN construction and search for valid paths is similar to prior work, we omit their detailed description and refer an interested reader to our technical report [41].

One difference worth mentioning, however, is that we use an *integer linear programming* (ILP) solver to find paths in the TTN, unlike prior approaches, which relied on SAT/SMT solvers. We found that although both solvers are equally quick at finding *one valid path*, when it comes to computing *all valid paths* of a given length, the ILP solver is much more efficient, as it has native support for enumerating multiple solutions.

```
1  \channel_name →                          \channel_name →
2    let x1 = c_list({});                     let x1 = c_list({});
3                                             x1' ← x1;
4    let x2 = x1.name;                        let x2 = x1'.name;
5    if x2 == channel_name;                   if x2 == channel_name;
6    let x3 = x1.id;                          let x3 = x1'.id;
7    let x4 = c_members(channel=x3);          let x4 = c_members(channel=x3);
8                                             x4' ← x4;
9    let x5 = u_info(user=x4);                let x5 = u_info(user=x4');
10   let x6 = x5.profile;                     let x6 = x5.profile;
11   let x7 = x6.email;                       let x7 = x6.email;
12                                            let x7' = return x7
13   x7                                       x7'
```

Figure 3.11. Array-oblivious program built from the bold path in Fig. 3.9 (left) and its lifted version (right).

Lifting array-oblivious programs. The function PROGS(π) (line 5 in Fig. 3.10) converts a TTN path π into a set of array-oblivious programs in A-Normal Form (ANF). Fig. 3.11 (left) shows the full array-oblivious program extracted from the bold path in Fig. 3.9. As you can see from this example, array-oblivious programs can be ill-typed: for example, the projection x_1`.name` in line 4 does not type-check since x_1 actually has an array type [`Channel`]. What we really want this program to do is to project `name` (and execute the remaining steps in the program) *for each* channel in x_1. This can be accomplished by inserting a monadic binding $x_1' \leftarrow x_1$ and using x_1' instead of x_1 in line 4 (and elsewhere in the program where a non-array version of x_1 is required, such as line 6). We refer to this process of repairing type errors by inserting monadic bindings and **return**s as *lifting*.[8]

The function LIFT (line 6 in Fig. 3.10) takes as input a semantic library $\hat{\Lambda}$, a query type $\hat{s}$, and an array-oblivious program $\mathcal{E}$, and produces a program $\mathcal{E}'$ that is well-typed at $\hat{s}$. Fig. 3.11 (right) depicts the result of lifting the program in Fig. 3.11 (left) to the query type `Channel.name` → [`Profile.email`] with $\hat{\Lambda}$ from Fig. 3.7. The full definition of lifting can be found in the technical report [41]. Informally, lifting type-checks the program "line by line", and whenever it encounters a type mismatch (in a projection, guard, or a method argument), it inserts

[8]A reader familiar with monads might think of the array-oblivious program as written in the identity monad instead of the list monad, and lifting as lifting the program back into the list monad.

the appropriate number of monadic bindings or `return`s in order to fix the mismatch. This is always possible because the only kind of type mismatch we can encounter is between an actual type $[..[\hat{t}]..]$ and the expected type $\hat{t}$, or vice versa. One thing worth noting is that we assume that the top-level return type of the program is an array type: since the lifted programs have top-level monadic bindings, they can only return arrays. If the user requests a scalar return type, we take this into account at the ranking stage by prioritizing programs that always return singleton arrays.

Completeness. Strictly speaking, array-oblivious search is incomplete: there are multiple programs that map to the same array-oblivious program, but lifting only returns a single, canonical representative. For example, consider the program in Fig. 3.11 (right), where we iterate over the array `x1` only once (line 3), and reuse the same “iterator" variable `x1'` in lines 4 and 6. An alternative would be to iterate over `x1` the second time before line 6, effectively retrieving names and IDs from all *pairs* of channels (instead of the name and the ID belonging to the same channel). We consider this a benign incompleteness because it is much less likely that the user intended to loop twice over the same array. If they did, we believe they would be able to repair the program by hand, as we discuss in Sec. 3.7.4.

3.6 Ranking

As we mentioned in Sec. 3.2, the algorithm SYNTHESIZE may generate hundreds or even thousands of well-typed candidate solutions, most of which, however, are uninteresting. We now formalize how APIPHANY ranks these candidates with the help of *retrospective execution* (RE).

Cost computation. To rank the programs, we assign them a positive cost, and then order them from lowest to highest cost. To compute the cost of a program $\mathcal{E}$, we retrospectively execute it multiple times, accumulating execution results in a set *res*; retrospective execution is non-deterministic, and executing a program more times lead to more precise cost estimates. We then compute the cost of $\mathcal{E}$ based on its result set *res* and the return type $\hat{t}$ of the query as follows:

1. The base cost is the size of $\mathcal{E}$ in AST nodes.

Retrospective Execution $\boxed{\langle \mathcal{W};\Gamma;\Sigma \mid e\rangle \Rightarrow v}$

$$\textsc{E-If-True-L}\ \frac{x_1 \in \Sigma \quad x_2 \notin \Sigma \quad \Sigma(x_1) = v_1 \qquad \langle \mathcal{W};\Gamma;x_2 \mapsto v_1,\Sigma \mid e\rangle \Rightarrow v}{\langle \mathcal{W};\Gamma;\Sigma \mid \texttt{if}\ x_1 = x_2; e\rangle \Rightarrow v}$$

$$\textsc{E-If-True-R}\ \frac{x_1 \notin \Sigma \quad \langle \mathcal{W};\Gamma;\Sigma \mid x_2\rangle \Rightarrow v_2 \qquad \langle \mathcal{W};\Gamma;x_1 \mapsto v_2, x_2 \mapsto v_2,\Sigma \mid e\rangle \Rightarrow v}{\langle \mathcal{W};\Gamma;\Sigma \mid \texttt{if}\ x_1 = x_2; e\rangle \Rightarrow v}$$

$$\textsc{E-Method-val}\ \frac{(f, \overline{l_i = v_i}, v_{out}) \in \mathcal{W}}{\langle \mathcal{W};\Gamma;\Sigma \mid f(\overline{l_i = v_i})\rangle \Rightarrow v_{out}}$$

$$\textsc{E-Method-name}\ \frac{\forall (f, \overline{l_i = v'_i}, v_{out}) \in \mathcal{W}.\ \exists i : v'_i \neq v_i \qquad (f, \overline{l_i = v'_i}, v_{out}) \in \mathcal{W}}{\langle \mathcal{W};\Gamma;\Sigma \mid f(\overline{l_i = v_i})\rangle \Rightarrow v_{out}}$$

Figure 3.12. Retrospective execution.

2. If $res = \emptyset$ (all executions have failed), the candidate receives a large penalty.

3. If $res = \{[]\}$ (all executions return an empty array), the candidate receives a medium penalty.

4. Finally, we compare the values $v \in res$ with the desired result type $\hat{t}$; recall that λ_A programs always return an array, while $\hat{t}$ might or might not be an array type. We assign a small penalty for a *multiplicity mismatch*, *i.e.* if either $\hat{t}$ is a scalar type and *any* value v has more than one element, or $\hat{t}$ is an array type and *all* values v have a single element.

Retrospective execution. We formalize RE as a judgement $\langle \mathcal{W};\Gamma;\Sigma \mid e\rangle \Rightarrow v$, stating that v is a valid result for executing the expression e in the environment Σ (which maps variables to values). The judgment is also parameterized by a type context Γ and witness set $\mathcal{W}$, used to replay method calls and sample program inputs. To run a candidate solution $\mathcal{E}$, we execute its body in an *empty environment* $\Sigma = \cdot$ and with Γ storing the types of $\mathcal{E}$'s arguments. As we explain in more detail below, program inputs are selected lazily, during execution, in order to maximize its chances of producing meaningful results.

Replaying method calls. Most of the rules for the RE judgement describe standard big-step operational semantics (they can be found in the technical report [41]), but two groups of rules, shown in Fig. 3.12, deserve more attention. The first group of interest includes E-METHOD-VAL and E-METHOD-NAME, which replay a method call by looking it up in $\mathcal{W}$. The rule E-METHOD-VAL applies when $\mathcal{W}$ contains an exact match for the current call, *i.e.* we have previously observed a call to the same method, with the same parameter names and parameter values. The rule E-METHOD-NAME applies when an exact match cannot be found (see first premise); in this case we pick an approximate match, where only the method name and parameter names match. Matching parameter names is important because many REST API methods admit optional parameters, and behave very differently based on which pattern of optional parameters is provided. If an approximate match cannot be found either, RE fails. Note that for a given call $f(\overline{l_i = v_i})$, there might be multiple approximate matches in $\mathcal{W}$, which makes RE non-deterministic (in fact, there can even be multiple precise matches because services are stateful). Due to hidden state and approximate matches, the results of RE are not guaranteed to match actual execution, but our experiments show that they are precise enough for the purposes of ranking.

Lazy sampling of program inputs. The remaining two rules in Fig. 3.12 are responsible for choosing program inputs so as to bias guard expressions to evaluate to true. We observe that when inputs are sampled eagerly ahead of time, guard expressions almost always evaluate to false, causing RE to return an empty array; as a result, our ranking heuristic cannot distinguish meaningful candidates from those that return an empty array regardless of the input. To address this issue, we postpone adding program inputs to the environment Σ until they are used. If the first usage of a program input is in a guard, the rules E-IF-TRUE-L and E-IF-TRUE-R pick its value to make the guard true: E-IF-TRUE-L applies when only the right-hand side of a guard is undefined, and E-IF-TRUE-R applies when the left-hand side or both are undefined. If the first usage of an input is in a method call or a projection, we instead randomly sample from all values of the same type observed in $\mathcal{W}$.

Table 3.1. APIs used in our experiments. For each API we report the number of methods $|\Lambda.f|$, min/max number of arguments per method n_{arg}, the number of objects $|\Lambda.o|$, and min/max size of the objects s_{obj}. We also report the number of witnesses $|\mathcal{W}|$ we collected for type mining and the number of methods covered by those witnesses n_{cov}.

	API size				API Analysis	
API	$\lvert\Lambda.f\rvert$	n_{arg}	$\lvert\Lambda.o\rvert$	s_{obj}	$\lvert\mathcal{W}\rvert$	n_{cov}
SLACK	174	0 - 15	79	1 - 70	3834	60
STRIPE	300	0 - 145	399	1 - 66	25402	124
SQUARE	175	0 - 20	716	1 - 34	1749	67

3.7 Evaluation

We implemented APIPHANY in Python, except for retrospective execution, where we used Rust for performance reasons. We used the Gurobi ILP solver [46] v9.1 as the back-end for TTN search. We ran all the experiments on a machine with an Intel Core i9-10850K CPU and 32GB of memory.

We designed our empirical evaluation to answer the following research questions:

(**RQ1**) Can APIPHANY find solutions for a wide range of realistic tasks across multiple popular APIs?

(**RQ2**) Is type mining effective and necessary for enabling type-directed synthesis?

(**RQ3**) Is retrospective execution effective and necessary for prioritizing relevant synthesis results?

API selection. For our evaluation, we selected three popular REST APIs: the SLACK communication platform and two online payment platforms, STRIPE and SQUARE. We selected these APIs because they are widely used and have both an OpenAPI specification and a web interface, which allowed us to set up the test environment and collect witnesses easily. As shown in Tab. 3.1, these APIs are quite complex: each has over a hundred methods with up to 145 arguments; all three feature optional arguments. The three APIs also contain a large number of object definitions, with up to 70 fields.

Experiment setup: type mining. Recall that type mining relies on a witness set $\mathcal{W}$. Witnesses are straightforward to collect for API owners, or when an integration test suite is publicly available; neither was the case in our setting. Instead, we collected witnesses by observing traffic from the services's web interface, and then enhancing this initial (very sparse) witness set via random testing; this process is described in more detail in our technical report [41]. As shown in Tab. 3.1, we collected between 1.7K and 25K witnesses per API, which covered 30–40% of all methods. It is hard to obtain full coverage for these closed source APIs as an outsider, for instance, because many methods are only available to paid accounts; our experiments show, however, that APIPHANY performs well with this witness set.

Benchmark selection. For each API, we extracted programming tasks from STACKOVERFLOW questions that mention this API as well as GITHUB repositories that use the API. After excluding the tasks that were out of scope of our DSL, we manually translated each of the remaining tasks from a natural-language description or a code snippet into a type query, resulting in 32 benchmarks (see Tab. 3.2). Apart from our running example (benchmark 1.1), these include, for instance: "Send a message to a user given their email" in SLACK (1.2), "Create a product and invoice a customer" in STRIPE (2.3), and "Delete catalog items with given names" in SQUARE (3.10). As noted in Tab. 3.2, many of these tasks are *effectful*: they require creating, modifying, or deleting objects.

Each benchmark comes with a "gold standard" solution: the accepted solution on STACKOVERFLOW or the snippet we found on GITHUB. We manually translated these solutions into APIPHANY's DSL. As shown in the "Solution Size" portion of Tab. 3.2, these solutions range in complexity from 7 to 22 AST nodes, containing up to three method calls and guards and up to seven projections, which makes them non-trivial for programmers to solve manually. A complete list of tasks, type queries, and solutions can be found in [41].

Experiment setup: program synthesis. For each of the 32 benchmarks, we ran the synthesizer with a timeout of 150 seconds. For each new candidate generated, we estimated its cost using

Table 3.2. Synthesis benchmarks and results. Benchmarks marked with † are effectful. For each benchmark we report the size of the desired solution: AST, n_f, n_p and n_g correspond to number of AST nodes, method calls, projections and guards, respectively. We also report the time to find the correct solution (in seconds), its rank without RE (r_{orig}), and the lower and upper bound on its rank with RE (r_{RE} and r_{RE}^{TO}). '-' means no solution is found in 150 seconds.

API	ID	Solution Size				Time	Rank		
		AST	n_f	n_p	n_g	(sec)	r_{orig}	r_{RE}	r_{RE}^{TO}
SLACK	1.1	17	3	6	1	83.5	25230	5	5
	1.2†	12	3	5	0	5.6	2224	10	10
	1.3	16	3	7	0	-	-	-	-
	1.4	14	2	4	1	1.3	489	24	31
	1.5†	10	2	3	0	3.4	788	5	5
	1.6†	9	2	2	0	1.7	573	8	19
	1.7†	12	2	4	1	1.3	757	8	9
	1.8	9	2	3	0	42.0	16438	29	30
STRIPE	2.1†	9	2	2	0	95.4	4952	3	3
	2.2†	10	2	2	0	92.4	4854	4	4
	2.3†	12	3	2	0	121.2	6363	1	1
	2.4	8	1	2	1	0.5	3	1	1
	2.5	8	2	2	0	1.0	10	4	4
	2.6†	9	3	2	0	12.2	270	3	3
	2.7	5	1	2	0	0.6	4	2	2
	2.8	16	2	7	1	20.2	679	17	17
	2.9	6	1	2	0	0.5	2	1	1
	2.10†	10	2	3	0	7.8	187	6	6
	2.11†	7	2	1	0	17.2	490	6	6
	2.12†	11	3	2	0	-	-	-	-
	2.13†	10	3	2	0	-	-	-	-
SQUARE	3.1	4	1	1	0	0.2	2	1	1
	3.2	16	1	4	3	0.5	10	4	4
	3.3	10	1	3	1	0.4	6	1	1
	3.4	5	1	2	0	0.7	2	1	1
	3.5†	14	2	3	0	2.2	99	2	2
	3.6	5	1	2	0	0.2	1	1	1
	3.7	6	1	2	0	0.3	7	4	4
	3.8	9	1	3	0	0.7	1	1	1
	3.9	8	1	2	1	0.2	3	2	2
	3.10†	16	2	5	1	1.9	174	10	12
	3.11†	8	2	3	0	1.0	68	16	16

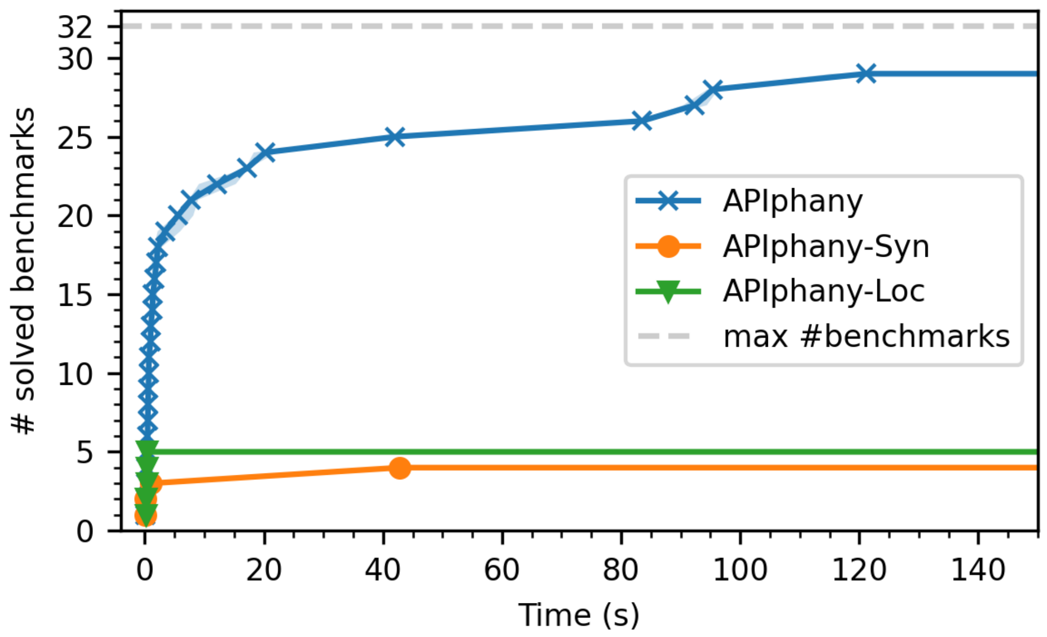


Figure 3.13. Comparison of synthesis performance between APIPHANY and its two variants that do not use type mining.

15 rounds of RE and recorded the synthesis time (including both TTN search and RE time). After the timeout, we checked whether the gold standard solution appears among the generated candidates and compared its RE-based rank vs the original rank at which it was generated (based on path length). Below we report average time and median rank over three runs to reduce the impact of randomness.

3.7.1 RQ1: Overall Effectiveness

The last four columns of Tab. 3.2 detail APIPHANY's performance on the 32 synthesis benchmarks. APIPHANY finds the correct solution for 29 benchmarks. The remaining three benchmarks fail with a timeout because their type queries are too ambiguous; for example, in benchmark 1.3 ("Get unread messages of a user") the type query has no means to specify that we are only interested in *unread* messages; as a result, the solution is drowned among thousands of other programs that map a user ID to messages.

We plot the number of benchmarks solved as a function of time (including RE) in

Fig. 3.13. As the plot shows, majority of benchmarks (19/32) can be solved within five seconds. On average APIPHANY takes 17.8 seconds to find the desired solution (median time 1.3 seconds).

Takeaway: APIPHANY is able to solve 91% of tasks from three real-world APIs.

3.7.2 RQ2: Type Mining

Recall that type mining involves replacing primitive *syntactic types* in the spec with unique *location-based types*, and then merging those based on the witness set to obtain *semantic types*. The merging process is not perfect: it might *fail to merge* two location that should have the same type because the witness set lacks evidence to justify the merge; or it might *spuriously merge* two locations if they share a value by chance. It is hard to measure the accuracy of inferred types directly, since we do not have an oracle for semantic types. Instead, we evaluate type mining indirectly in two ways: 1) we run an *ablation study* to measure its impact on the overall performance of the synthesizer, and 2) we perform a small-scale *qualitative analysis* of inferred types.

Ablation study. For this experiment, we compare the performance of APIPHANY and its two variants: (a) APIPHANY-SYN, which builds the TTN directly from syntactic types, and (b) APIPHANY-LOC, which builds the TTN from (unmerged) location-based types. We plot the number of benchmarks solved by each variant as the function of time in Fig. 3.13.

As expected, both variants perform poorly: APIPHANY-SYN only solves 4/32 benchmarks and APIPHANY-LOC solves 5. All these benchmarks are "easy" (solved by APIPHANY in under a second). Intuitively, the two variants represent two extremes in terms of type *granularity*. Syntactic types are *too coarse-grained* (all `String` locations have the same type), which leads TTN search to return too many well-typed candidates. As a result, APIPHANY-SYN struggles to solve all but the simplest tasks, with many benchmarks running out of memory. Location-based types, on the other hand, are *too fine-grained* (each `String` location has a unique type), which leads to most desired solutions simply being ill-typed, because there is no way for one method to use values returned by another. The solutions to all of the five benchmarks solved by APIPHANY-LOC have only one method call with no parameters, followed by several projections or filters.

As you can see from Fig. 3.13, APIPHANY drastically outperforms both variants. This result indicates that type mining strikes a good balance between coarse- and fine-grained types: all 32 benchmarks have a well-typed solution in terms of the mined types, and APIPHANY is able to find most of them within a reasonable time.

Qualitative analysis. To give a more direct account of the quality of inferred semantic types, we randomly sampled five methods from each API (among the methods covered by the collected witnesses), and manually inspected the inferred types to check if they match our expectations. More specifically, for each `String` location in a method spec, we pick a location type loc^*, which we deem most natural for a programmer to use in a type query (for example, for the parameter to `users_info`, $loc^* =$ `User.id`); we consider the inferred loc-set type sufficient if it contains loc^*. The detailed results appear in the technical report [41].

In the methods we examined, type mining was able to infer a sufficient semantic type for all responses, required parameters, and about half of optional parameters. The remaining optional parameters were assigned unmerged location types, because they were never used in our witness set. This is almost unavoidable, because of the sheer number of obscure optional parameters in real-world APIs (which, fortunately, are rarely needed to solve programmer's tasks).

Recall that the other failure mode of type mining is spuriously merging unrelated locations. We did not observe any spurious merges among the randomly sampled methods, but anecdotally we did encounter one such merge elsewhere in the Slack API: between `Channel.name` and `Message.name`. Note that spurious merges might slow down the search and produce some "semantically ill-typed" solutions, but they do not prevent APIPHANY from finding the desired solution.

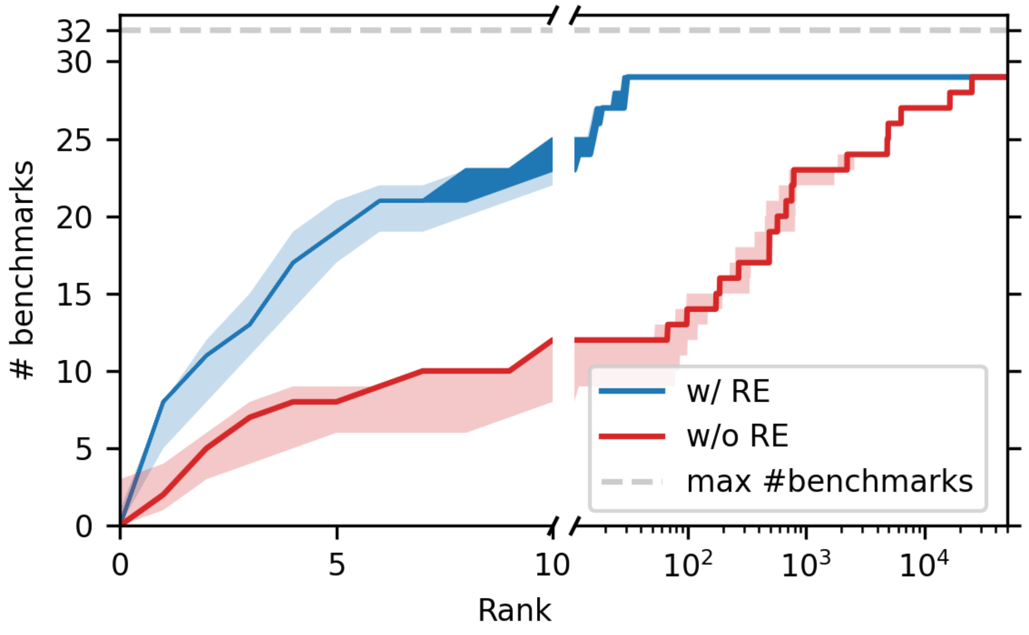


Figure 3.14. Number of benchmarks whose solution is reported within a given rank. The filled blue area is the range of ranks one might get depending on when they inspect the candidates. The shaded area is the 95% confidence interval.

Takeaway: Type mining increases the percentage of solved benchmarks from 12% to 91%.

3.7.3 RQ3: Ranking

To measure the effectiveness of RE-based ranking, we compare the last three columns of Tab. 3.2: r_{orig} denotes the rank of the desired solution in the order it was generated by TTN search (which is based on path length, and hence correlated with solution size); r_{RE} denotes the RE-based rank of the solution at the time it was generated, and r_{RE}^{TO} denotes its RE-based rank by the timeout (which can be lower than r_{RE} as other candidates generated later might end up being ranked higher). We report both of these RE-based ranks because we envision an APIPHANY user inspecting the candidate solutions some time between they are generated and the timeout, and hence the relevant rank value is between r_{RE} and r_{RE}^{TO}. We plot the number of benchmarks whose solutions lie at or below each rank in Fig. 3.14, with the range between r_{RE} and r_{RE}^{TO} represented as a filled area.

As you can see from Fig. 3.14, RE-based ranking significantly increases the chances that the desired solution makes the short-list of candidates. In particular, *without RE-based ranking* only 8 benchmarks (28% of solved) return the correct solution in top five, and only 12 (41%) return it in top ten; in contrast, *with RE-based ranking*, 19 (65%) benchmarks return the correct solution in top five (after timeout), and 23 (79%) in top ten. Moreover, as we can see from Tab. 3.2, the solution's rank never gets worse after RE, in all but two cases it strictly improves, and for all long-running benchmarks it improves drastically (the average rank improves from 2230.5 to 7.0).

A closer look at the six benchmarks that do not land in top ten after RE reveals two main reasons for these suboptimal rankings. In most cases the solution is simply large, and there are many smaller candidates that are still meaningful. For example, the query "Delete all catalog items" (3.11) takes no arguments and returns an array of all deleted items; there are many valid and simple ways to construct an array of catalog items without deleting them. In a few cases, APIphany fails to throw out meaningless programs due to the imprecision of retrospective execution. For example, in 1.6 it reports a solution that posts an update to a given channel with a given timestamp, even though this timestamp might be invalid for this channel; APIphany instead thinks that this call always succeeds by relying on approximate matches during retrospective execution.

We also recorded the time APIPHANY takes to compute the cost for all generated candidates (which involves executing each candidate 15 times). Although APIPHANY generates thousands of well-typed candidates for most benchmarks, cost computation only takes about 1% of total synthesis time.

Takeaway: RE-based ranking takes a negligible amount of time and increases the percentage of correct solutions reported in top ten from 41% to 79%.

3.7.4 Discussion and Limitations

Witness generation. One threat to validity of our evaluation is that the results of type minings (and therefore synthesis) depend heavily on the witness set. In particular, if our benchmarks required methods that are not covered by the witness set, APIPHANY most likely would not be able to solve them, since they would be ill-typed with inferred semantic types. We ran our experiments using a particular witness set, which we collected using one methodology (described in the technical report [41]); our findings might not generalize to using APIPHANY with witness sets collected by other means.

Effectful methods. We observe that effectful methods in REST APIs have an interesting property: they make the effect explicit in their response. For example, the method for posting a message on SLACK also returns the message object, and the method for deleting a catalog item in SQUARE returns the ID of the deleted item (instead of just returning `void`). This property makes REST APIs particularly suitable for type-directed synthesis and expressing user intent with types: for example, the query "Send a message to a user with a given email" can be expressed as the type `Profile.email` $\rightarrow$ `Message` instead of a much less informative type `Profile.email` $\rightarrow$ `void`. The downside, of course, is that the return type of an effectful method might not be obvious to the user (for example, does deleting a catalog item return an object or its ID?) One way to overcome this limitation is to let the user specify the name of the last method they want to call (*e.g.* `catalog_object_delete`) instead of the output type; this kind of specification is straightforward to integrate into TTN search.

DSL restrictions. In our search for benchmarks, we encountered (very few) snippets that were inexpressible in our DSL because they required functional transformations on primitive values, as opposed to just structural transformations on objects and arrays, for example: "Get all members of a channel and *concatenate* them together". We consider such functional transformations beyond the scope of APIPHANY because its type-based specifications are too coarse to distinguish between different functional transformations. This is also the reasoning behind our design decision to only support equality inside guards, as opposed to more general predicates: if the specification cannot distinguish between, say, $=$ and $\leq$, there is little use in generating programs with both. More generally, we view programs synthesized by APIPHANY as a starting point, which helps the programmer figure out how to plumb data through a set of API calls; we envision the user building on top of those programs to add functional modifications and more expressive predicates. This interaction model motivates both our DSL restrictions and our type-based specifications.

Value-based location merging. Value-based merging works well for strings, since their large domain makes it unlikely that two `String` locations share a value by chance. It works less well for other primitive types, such as integers and booleans. To reduce the risk of spurious merges, our implementation performs value-based merging only for strings and large integers (> 1000), but not for booleans or small integers. In the future, we plan to investigate more sophisticated approaches to location merging. One idea is to use probabilistic reasoning to estimate the likelihood of two locations having the same type based on (1) how common a value is across locations and (2) what proportion of values is shared between the two locations. Another approach is to cluster locations using NLP techniques, such as sentiment analysis of object and field names, as well as documentation.

User interface. Another important direction for future work is to investigate usable ways of specifying semantic type queries and comprehending synthesis results. In particular, existing work from the HCI community [36, 37] might help users quickly explore a large space of related candidate solutions, thereby mitigating the limitations of ranking.

3.8 Related Work

APIPHANY is a component-based synthesizer and primarily compares with related work in this space. It also draws on techniques from specification mining and type inference.

Type-directed component-based synthesis. The goal of component-based synthesis is to find a *composition* of components (library functions) that implements a given task. In *type-directed* component-based synthesis both the task and the components are specified using types. The traditional approach to this problem based on proof search [5, 74, 48] scales poorly with the size of the component library. An alternative, more scalable *graph-based approach* was introduced in PROSPECTOR [64] for unary components, and generalized to *n*-ary components in SYPET [27], by replacing graphs with Petri nets. TYGAR [45] further extends SYPET's search to polymorphic components using the idea of *abstract types*, which are inspired by *succinct types* from another component-based synthesizer, INSYNTH [47]. APIPHANY's program search phase is using the Petri net encoding from SYPET and TYGAR with minor adaptations (support for optional arguments and ILP encoding). Our array-oblivious encoding is related to abstract and succinct types in that it helps make the Petri net smaller, but it is also substantially different in that, unlike prior work, it can efficiently encode a certain class of higher-order programs (array comprehensions) into the Petri net.

API navigation. Beyond type-directed synthesis, other work focuses on auto-completion [77, 86, 63] but relies on static analysis and mining client code, which APIPHANY does not require. Among tools that leverage dynamic analysis, EDSYNTH [108] uses test executions to generate snippets that involve both API calls and control structures. MATCHMAKER [110] and DEMOMATCH [109] are similar to APIPHANY in that they rely on observed program traces to suggest code that uses complex APIs (the former from types and the latter from demonstrations). All these techniques work in the context of Java, and hence assume that sufficiently precise types are already present.

SQL synthesis. The problem of generating projections and filters is related to synthesis of SQL queries [101, 106]. Existing SQL synthesis techniques are not directly applicable to our problem domain, because (1) our programs also contain arbitrary API method invocations, and (2) we manipulate semi-structured data instead of relational data.

API discovery and specification mining. A complimentary approach to API navigation using program synthesis is to infer specifications [3, 90, 67] or example usages [50, 15, 10] to help the user understand the API better. APIPHANY's type mining is inspired by [3], where they build probabilistic finite state automata representing data and temporal dependencies between API methods. APIPHANY implements a simpler form of their algorithm, which discovers data flows (but not temporal dependencies), but the novelty lies in using this information to drive program synthesis.

Type mining is also related to prior work on inferring type annotations for dynamically typed languages from executions [16, 4, 13]. However, this work is for structural types, whereas we infer domain-specific nominal types.

Simulated execution. An alternative to our retrospective execution is to synthesize a *model* of the API, and evaluate program candidates against that model. Previous work [49, 53] synthesizes models for complex frameworks and opaque code; our retrospective execution is simpler: it skips the extra step of model synthesis.

Ranking solutions. Specifications in program synthesis are often ambiguous, so synthesizers have to rank their candidate solutions and return the top result(s). Existing tools most commonly rely on hand-crafted [39] or learned [47, 86, 92] ranking functions based on syntactic features of generated programs. HOOGLE+ [52] is most similar to APIPHANY in that it ranks programs based on the results of their *execution*, using heuristics like whether the program always fails, and how similar it is to other candidates.

3.9 Acknowledgements

This chapter, in part, is a reprint of the material as it appears in Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, Nadia Polikarpova. ACM 2022. The dissertation author was a principal author and investigator of this paper.

Conclusion

This dissertation has introduced type-directed techniques applied to component-based synthesis, tackling the API discovery challenge in the context of polymorphic types and more general types.

For polymorphic types in functional programming languages, this dissertation proposes two approaches. The first approach involves applying overapproximation to polymorphic datatypes. Based on abstract types, we construct an abstract type transition graph and use the graph reachability algorithm to find programs. It is important to note that while these programs are guaranteed to type check in the abstract type system, there is no assurance of type checking in the concrete type system. If the algorithm yields an ill-typed term in the concrete type system, refinement of the type abstraction is necessary to exclude such terms. This refinement process continues until the generated program is well-typed in both the abstract and concrete type systems. However, with each change in type abstraction, the abstract type transition graph needs to be reconstructed. Compared to previous methods, this algorithm loses the advantage of precomputing the type transition graph, and the reconstruction of the abstract type transition graph introduces significant overhead to the overall synthesis time.

As a solution to such significant overhead time, this dissertation proposes equality-constrained tree automata (ECTAs) as a novel representation of the search space. ECTAs augment tree automata with equality constraints, enhancing the expressiveness of tree automata to compactly represent entangled terms. Alongside this representation, an enumeration algorithm is introduced to efficiently retrieve terms from the compact representation. ECTAs successfully reduce the implementation size of a type-directed synthesizer and accelerate the search speed compared with prior work. It is worth noting that type-directed synthesis is just one of the potential applications of this data structure. More interesting applications such as SAT solvers, specification mining, database optimization, *etc* can be studied. As part of the contributions of this dissertation, we have published this data structure as a Haskell library online.

In addition to polymorphic types, this dissertation investigates type-directed synthesis in the context of more general types such as `int`, `string`, *etc*. These general types are too coarse-grained for effective program synthesis, as they allow for a large number of well-typed programs. To address this issue, this work proposes a semantic type system along with a type inference algorithm, allowing expressing user intent more precisely while ruling out unreasonable programs. The inference algorithm automatically discovers semantic types for components from their execution traces based on value coincidances. This type system is proven to be effective in assigning semantic types to components and the inferred types successfully guide the synthesizer to find desired programs.

Although the proposed techniques can effectively solve real-world tasks, they are typically limited to relatively small-sized tasks. They suffer from the notorious scalability problem and struggle to tackle tasks that require solutions consisting of a large number of lines of code. With the emergence of large language models (LLMs), which excel at finding large-scale program sketches but often have difficulty in filling in details, an interesting future direction could be to integrate type-directed techniques with LLMs. LLMs address the scalability issue, while type-directed search ensures the correctness of detailed program implementations. This integration could lead to more robust and scalable solutions for complex programming tasks.

Bibliography

[1] Michael D Adams and Matthew Might. Restricting grammars with tree automata. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–25, 2017.

[2] Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. Scaling enumerative program synthesis via divide and conquer. In *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Part I*, pages 319–336, 2017.

[3] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 4–16, New York, NY, USA, 2002. Association for Computing Machinery.

[4] Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for ruby. In Thomas Ball and Mooly Sagiv, editors, *POPL. Austin, TX, USA, January 26-28, 2011*, pages 459–472. ACM, 2011.

[5] Lennart Augusstson. Djinn. https://github.com/augustss/djinn, 2005.

[6] Leo Bachmair and Nachum Dershowitz. Equational Inference, Canonical Proofs, and Proof Orderings. *Journal of the ACM (JACM)*, 41(2):236–276, 1994.

[7] Hendrik Pieter Barendregt. *The lambda calculus - its syntax and semantics*, volume 103 of *Studies in logic and the foundations of mathematics*. North-Holland, 1985.

[8] Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. The Emptiness Problem for Tree Automata with Global Constraints. In *2010 25th Annual IEEE Symposium on Logic in Computer Science*, pages 263–272. IEEE, 2010.

[9] Luis Barguñó, Carles Creus, Guillem Godoy, Florent Jacquemard, and Camille Vacher. Decidable classes of tree automata mixing local and global constraints modulo flat theories. *Logical Methods in Computer Science*, 9, 02 2013.

[10] Celeste Barnaby, Koushik Sen, Tianyi Zhang, Elena Glassman, and Satish Chandra. Exempla gratis (e.g.): Code examples for free. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of*

Software Engineering, ESEC/FSE 2020, pages 1353–1364, New York, NY, USA, 2020. Association for Computing Machinery.

[11] Bruno Bogaert, Franck Seynhaeve, and Sophie Tison. The Recognizability Problem for Tree Automata with Comparisons Between Brothers. In *International Conference on Foundations of Software Science and Computation Structure*, pages 150–164. Springer, 1999.

[12] Bruno Bogaert and Sophie Tison. Equality and Disequality Constraints on Direct Subterms in Tree Automata. In *Annual Symposium on Theoretical Aspects of Computer Science*, pages 159–171. Springer, 1992.

[13] Ambrose Bonnaire-Sergeant. *Typed Clojure in Theory and Practice*. PhD thesis, Indiana University, Bloomington, 2019.

[14] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 213–222, New York, NY, USA, 2009. Association for Computing Machinery.

[15] Raymond P. L. Buse and Westley Weimer. Synthesizing api usage examples. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 782–792. IEEE Press, 2012.

[16] Ravi Chugh, Sorin Lerner, and Ranjit Jhala. Type inference with run-time logs. In *Workshop on Scripts to Programs (STOP)*, 2011.

[17] Edmund M. Clarke, Robert P. Kurshan, and Helmut Veith. The localization reduction and counterexample-guided abstraction refinement. In *Time for Verification, Essays in Memory of Amir Pnueli*, pages 61–71, 2010.

[18] Patrick Cousot. Types as abstract interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 316–331, New York, NY, USA, 1997. ACM.

[19] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM.

[20] Max Dauchet. Rewriting and Tree Automata. In *French School on Theoretical Computer Science*, pages 95–113. Springer, 1993.

[21] Max Dauchet, Anne-Cécile Caron, and Jean-Luc Coquidé. Automata for reduction properties solving. *Journal of Symbolic Computation*, 20(2):215–233, 1995.

[22] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[23] David Detlefs, Greg Nelson, and James B Saxe. Simplify: A Theorem Prover for Program Checking. *Journal of the ACM (JACM)*, 52(3):365–473, 2005.

[24] Roberto Di Cosmo. Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming*, 3(3):485–525, 1993. Special Issue on ML.

[25] Roy Dyckhoff and Luís Pinto. Proof search in constructive logics. In *In Sets and proofs*, pages 53–65. Cambridge University Press, 1998.

[26] Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp J. Meyer, and Filip Niksic. An smt-based approach to coverability analysis. In *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, pages 603–619, 2014.

[27] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. Component-based synthesis for complex apis. In *POPL*, 2017.

[28] Kasra Ferdowsifard, Shraddha Barke, Hila Peleg, Sorin Lerner, and Nadia Polikarpova. Loopy: Interactive program synthesis with control structures. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[29] Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability Analysis over Term Rewriting Systems. *Journal of Automated Reasoning*, 33(3):341–383, 2004.

[30] Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. Example-directed synthesis: a type-theoretic interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 802–815, 2016.

[31] Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. Codehint: Dynamic and interactive synthesis of code snippets. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 653–663, New York, NY, USA, 2014. ACM.

[32] Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. From many places to few: Automatic abstraction refinement for petri nets. In *Petri Nets and Other Models of Concurrency - ICATPN 2007, 28th International Conference on Applications and Theory of Petri Nets and Other Models of Concurrency, ICATPN 2007, Siedlce, Poland, June 25-29, 2007, Proceedings*, pages 124–143, 2007.

[33] Ronald Garcia, Alison M. Clark, and Éric Tanter. Abstracting gradual typing. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 429–442, 2016.

[34] Alfons Geser, Dieter Hofbauer, Johannes Waldmann, and Hans Zantema. On Tree Automata that Certify Termination of Left-Linear Term Rewriting Systems. *Information and Computation*, 205(4):512–534, 2007.

[35] Matthías Páll Gissurarson. Suggesting valid hole fits for typed-holes (experience report). In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell*, Haskell 2018, pages 179–185. Association for Computing Machinery, 2018.

[36] Elena L. Glassman, Jeremy Scott, Rishabh Singh, Philip J. Guo, and Robert C. Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Trans. Comput.-Hum. Interact.*, 22(2), mar 2015.

[37] Elena L. Glassman, Tianyi Zhang, Björn Hartmann, and Miryung Kim. *Visualizing API Usage Examples at Scale*, pages 1–12. Association for Computing Machinery, New York, NY, USA, 2018.

[38] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with pvs. In *Computer Aided Verification*, pages 72–83, 1997.

[39] Sumit Gulwani. Automating string processing in spreadsheets using input-output examples. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 317–330, 2011.

[40] Sumit Gulwani. Automating String Processing in Spreadsheets using Input-Output Examples. *ACM Sigplan Notices*, 46(1):317–330, 2011.

[41] Zheng Guo, David Cao, Davin Tjong, Jean Yang, Cole Schlesinger, and Nadia Polikarpova. Type-directed program synthesis for restful apis (technical report). *arXiv preprint arXiv:2203.16697*, 2022.

[42] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement. *Proc. ACM Program. Lang.*, 4(POPL), dec 2019.

[43] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. Program synthesis by type-guided abstraction refinement, 2019.

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

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

[46] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2021.

[47] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. Complete completion using types and weights. In *PLDI*, 2013.

[48] George T. Heineman, Jan Bessai, Boris Düdder, and Jakob Rehof. 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*, pages 303–317, 2016.

[49] Stefan Heule, Manu Sridharan, and Satish Chandra. Mimic: Computing models for opaque code. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 710–720, New York, NY, USA, 2015. Association for Computing Machinery.

[50] Abbas Heydarnoori, Krzysztof Czarnecki, and Thiago Tonelli Bartolomei. Supporting framework use via automatically extracted concept-implementation templates. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 344–368, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[52] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. Digging for fold: synthesis-aided API discovery for haskell. *Proc. ACM Program. Lang.*, 4(OOPSLA):205:1–205:27, 2020.

[53] Jinseong Jeon, Xiaokang Qiu, Jonathan Fetter-Degges, Jeffrey S. Foster, and Armando Solar-Lezama. Synthesizing framework models for symbolic execution. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 156–167, New York, NY, USA, 2016. Association for Computing Machinery.

[54] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 215–224, New York, NY, USA, 2010. Association for Computing Machinery.

[55] Susumu Katayama. An analytical inductive functional programming system that avoids unintended programs. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation, PEPM 2012, Philadelphia, Pennsylvania, USA, January 23-24, 2012*, pages 43–52, 2012.

[56] Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. Incremental, inductive coverability. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, pages 158–173, 2013.

[57] Donald E Knuth. Semantics of Context-Free Languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.

[58] James Koppel. Version Space Algebras are Acyclic Tree Automata, 2021.

[59] James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. Searching entangled program spaces (extended version), 2022.

[60] M W Krentel. The complexity of optimization problems. In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, STOC '86, page 69–76, New York, NY, USA, 1986. Association for Computing Machinery.

[61] Tessa Lau, Steven A Wolfman, Pedro Domingos, and Daniel S Weld. Programming by Demonstration Using Version Space Algebra. *Machine Learning*, 53(1):111–156, 2003.

[62] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. Accelerating search-based program synthesis using learned probabilistic models. In *PLDI*, 2018.

[63] Sifei Luan, Di Yang, Celeste Barnaby, Koushik Sen, and Satish Chandra. Aroma: Code recommendation via structural code search. *Proc. ACM Program. Lang.*, 3(OOPSLA), October 2019.

[64] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *PLDI*, 2005.

[65] Erik Meijer, Brian Beckman, and Gavin Bierman. Linq: Reconciling object, relations and xml in the .net framework. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, page 706, New York, NY, USA, 2006. Association for Computing Machinery.

[66] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 997–1016, New York, NY, USA, 2012. Association for Computing Machinery.

[67] Alon Mishne, Sharon Shoham, and Eran Yahav. Typestate-based semantic code search over partial programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 997–1016, New York, NY, USA, 2012. Association for Computing Machinery.

[68] Neil Mitchell. Hoogle. https://www.haskell.org/hoogle/, 2004.

[69] Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. Bayesian specification learning for finding API usage errors. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*, pages 151–162, 2017.

[70] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–44, 2020.

[71] Chandrakana Nandi, Max Willsey, Amy Zhu, Yisu Remy Wang, Brett Saiki, Adam Anderson, Adriana Schulz, Dan Grossman, and Zachary Tatlock. Rewrite rule inference using equality saturation. *Proc. ACM Program. Lang.*, 5(OOPSLA), oct 2021.

[72] Greg Nelson and Derek C. Oppen. Fast Decision Procedures Based on Congruence Closure. *J. ACM*, 27(2):356–364, 1980.

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

[74] Ulf Norell. Dependently typed programming in agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, pages 230–266, 2008.

[75] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 619–630, 2015.

[76] Jukka Paakki. Attribute Grammar Paradigms—A High-Level Methodology in Language Implementation. *ACM Computing Surveys (CSUR)*, 27(2):196–255, 1995.

[77] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, pages 275–286, 2012.

[78] Benjamin C. Pierce. *Advanced Topics in Types and Programming Languages*. The MIT Press, 2004.

[79] Benjamin C. Pierce and David N. Turner. Local type inference. *ACM Trans. Program. Lang. Syst.*, 22(1):1–44, 2000.

[80] Gordon Plotkin. *Lattice Theoretic Properties of Subsumption*. Edinburgh University, Department of Machine Intelligence and Perception, 1970.

[81] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. Program synthesis from polymorphic refinement types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016*, pages 522–538, 2016.

[82] Joshua Pollock and Altan Haan. E-Graphs Are Minimal Deterministic Finite Tree Automata (DFTAs) · Discussion #104 · egraphs-good/egg, 2021.

[83] Oleksandr Polozov and Sumit Gulwani. FlashMeta: A Framework for Inductive Program Synthesis. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 107–126, 2015.

[84] Inc. Postman. 2020 State of The API Report. https://www.postman.com/state-of-api/api-technologies/, 2020.

[85] Varot Premtoon, James Koppel, and Armando Solar-Lezama. Semantic Code Search via Equational Reasoning. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 1066–1082, 2020.

[86] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. *SIGPLAN Not.*, 49(6):419–428, June 2014.

[87] Andreas Reuß and Helmut Seidl. Bottom-up tree automata with term constraints. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 581–593. Springer, 2010.

[88] Ilya Sergey, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Joachim Breitner. Modular, higher order cardinality analysis in theory and practice. *J. Funct. Program.*, 27:e11, 2017.

[89] Kensen Shi, Jacob Steinhardt, and Percy Liang. Frangel: Component-based synthesis with control structures. *Proc. ACM Program. Lang.*, 3(POPL):73:1–73:29, January 2019.

[90] Sharon Shoham, Eran Yahav, Stephen Fink, and Marco Pistoia. Static specification mining using automata-based abstractions. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2007, London, UK, July 9-12, 2007*, pages 174–184, 2007.

[91] Jeremy G. Siek and Walid Taha. Gradual typing for functional languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*, pages 81–92, 2006.

[92] Rishabh Singh and Sumit Gulwani. Predicting a correct program in programming by example. In *CAV - 27th International Conference, 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part I*, pages 398–414, 2015.

[93] Inc. SmartBear. The State of API Report 2020. https://static1.smartbear.co/smartbearbrand/media/pdf/smartbear_state_of_api_2020.pdf, 2020.

[94] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, Berkeley, CA, USA, 2008. AAI3353225.

[95] Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. From program verification to program synthesis. In *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 313–326, 2010.

[96] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, April 1975.

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

[98] Pawel Urzyczyn. 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*, pages 373–389, 1997.

[99] Eric Van Wyk, Derek Bodin, Jimin Gao, and Lijesh Krishnan. Silver: An Extensible Attribute Grammar System. *Science of Computer Programming*, 75(1-2):39–54, 2010.

[100] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76, 1989.

[101] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive sql queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 452–466, New York, NY, USA, 2017. Association for Computing Machinery.

[102] Xinyu Wang, Isil Dillig, and Rishabh Singh. Synthesis of Data Completion Scripts using Finite Tree Automata. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–26, 2017.

[103] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *PACMPL*, 2(POPL), 2018.

[104] Xinyu Wang, Isil Dillig, and Rishabh Singh. Program synthesis using abstraction refinement. *Proc. ACM Program. Lang.*, 2(POPL):63:1–63:30, 2018.

[105] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. Egg: Fast and extensible equality saturation. *Proceedings of the ACM on Programming Languages*, 5(POPL):1–29, 2021.

[106] Navid Yaghmazadeh, Yuepeng Wang, Isil Dillig, and Thomas Dillig. Sqlizer: Query synthesis from natural language. *Proc. ACM Program. Lang.*, 1(OOPSLA), October 2017.

[107] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. In A. Smola, A. Dimakis, and I. Stoica, editors, *Proceedings of Machine Learning and Systems*, volume 3, pages 255–268, 2021.

[108] Z. Yang, J. Hua, K. Wang, and S. Khurshid. Edsynth: Synthesizing api sequences with conditionals and loops. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 161–171, 2018.

[109] Kuat Yessenov, Ivan Kuraj, and Armando Solar-Lezama. Demomatch: Api discovery from demonstrations. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 64–78, New York, NY, USA, 2017. Association for Computing Machinery.

[110] Kuat Yessenov, Zhilei Xu, and Armando Solar-Lezama. Data-driven synthesis for object-oriented frameworks. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '11, pages 65–82, New York, NY, USA, 2011. Association for Computing Machinery.