Program Synthesis by Type-Guided Abstraction Refinement

ZHENG GUO, UC San Diego, USA
MICHAEL JAMES, UC San Diego, USA
DAVID JUSTO, UC San Diego, USA
JIAXIAO ZHOU, UC San Diego, USA
ZITENG WANG, UC San Diego, USA
RANJIT JHALA, UC San Diego, USA
NADIA POLIKARPOVA, UC San Diego, USA

We consider the problem of type-directed component-based synthesis where, given a set of (typed) components and a query *type*, the goal is to synthesize a *term* that inhabits the query. Classical approaches based on proof search in intuitionistic logics do not scale up to the standard libraries of modern languages, which span hundreds or thousands of components. Recent graph reachability based methods proposed for Java do scale, but only apply to monomorphic data and components: polymorphic data and components infinitely explode the size of the graph that must be searched, rendering synthesis intractable. We introduce *type-guided abstraction refinement* (TYGAR), a new approach for scalable type-directed synthesis over polymorphic datatypes and components. Our key insight is that we can overcome the explosion by building a graph over *abstract types* which represent a potentially unbounded set of concrete types. We show how to use graph reachability to search for candidate terms over abstract types, and introduce a new algorithm that uses *proofs of untypeability* of ill-typed candidates to iteratively *refine* the abstraction until a well-typed result is found.

We have implemented TYGAR in H+, a tool that takes as input a set of Haskell libraries and a query type, and returns a Haskell term that uses functions from the provided libraries to implement the query type. Our support for polymorphism allows H+ to work with higher-order functions and type classes, and enables more precise queries due to parametricity. We have evaluated H+ on 44 queries using a set of popular Haskell libraries with a total of 291 components. H+ returns an interesting solution within the first five results for 32 out of 44 queries. Our results show that TYGAR allows H+ to rapidly return well-typed terms, with the median time to first solution of just 1.4 seconds. Moreover, we observe that gains from iterative refinement over exhaustive enumeration are more pronounced on harder queries.

12

CCS Concepts: • **Theory of computation → Automated reasoning**; • **Software and its engineering → Automatic programming**.

Additional Key Words and Phrases: Program Synthesis, Type Systems, Abstract Interpretation

ACM Reference Format:
Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2020. Program Synthesis by Type-Guided Abstraction Refinement. *Proc. ACM Program. Lang.* 4, POPL, Article 12 (January 2020), 28 pages. https://doi.org/10.1145/3371080

Authors' addresses: Zheng Guo, UC San Diego, USA, zhg069@ucsd.edu; Michael James, UC San Diego, USA, m3james@ucsd.edu; David Justo, UC San Diego, USA, djusto@ucsd.edu; Jiaxiao Zhou, UC San Diego, USA, jiz417@ucsd.edu; Ziteng Wang, UC San Diego, USA, ziw329@ucsd.edu; Ranjit Jhala, UC San Diego, USA, jhala@cs.ucsd.edu; Nadia Polikarpova, UC San Diego, USA, npolikarpova@ucsd.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
© 2020 Copyright held by the owner/author(s).
2475-1421/2020/1-ART12
https://doi.org/10.1145/3371080

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 [Mitchell 2004], 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 [Urzyczyn 1997]. Consequently, one approach is to develop synthesizers based on proof search in intuitionistic logics [Augusstson 2005]. 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 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 [Mandelin et al. 2005]. 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 [Feng et al. 2017].

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. 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 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*

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

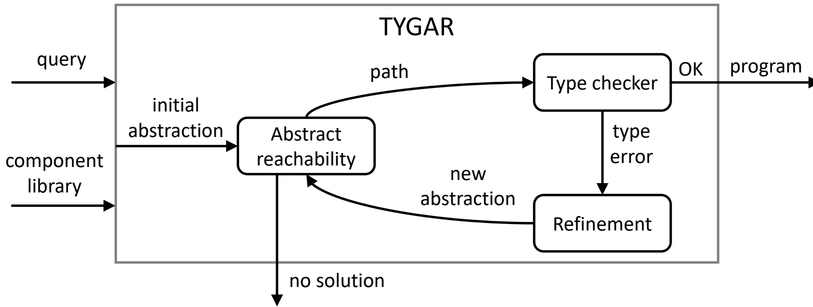

Fig. 1. Overview of the TYGAR synthesis algorithm.

refinement (SYNGAR) [Wang et al. 2018], 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 [Wadler and Blott 1989], which again, relies crucially on support for parametric polymorphism.

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

1. Abstract Typing. Our first contribution is a novel notion of *abstract typing* grounded in the framework of abstract interpretation [Cousot and Cousot 1977]. 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 [Graf and Saidi 1997], 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 [Feng et al. 2017; Shi et al. 2019], 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* [Sergey et al. 2017] 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

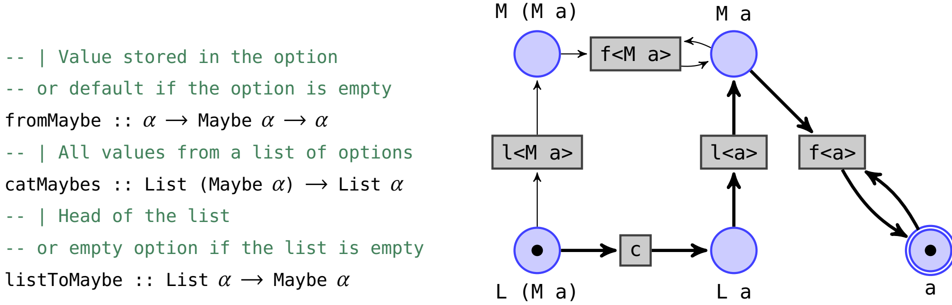

Fig. 2. (left) A tiny component library. (right) A Type Transition Net for this library and query a ⟶ List (Maybe a) ⟶ a. The transitions l<a>, f<a> (resp. l<M a>, f<M a>) correspond to the polymorphic instances of the components listToMaybe, fromMaybe at type a (resp. M a).

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 BACKGROUND AND OVERVIEW

We start with some examples that illustrate the prior work on component-based synthesis that H+ builds on (Sec. 2.1), the challenges posed by polymorphic components, and our novel techniques for addressing those challenges.

2.1 Synthesis via Type Transition Nets

The starting point of our work is SyPet [Feng et al. 2017], 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. 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. 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

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

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. 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. 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 [`c`, `l`, `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.

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. 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

[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 [Cousot and Cousot 1977], *i.e.* an abstraction of a set of concrete types.

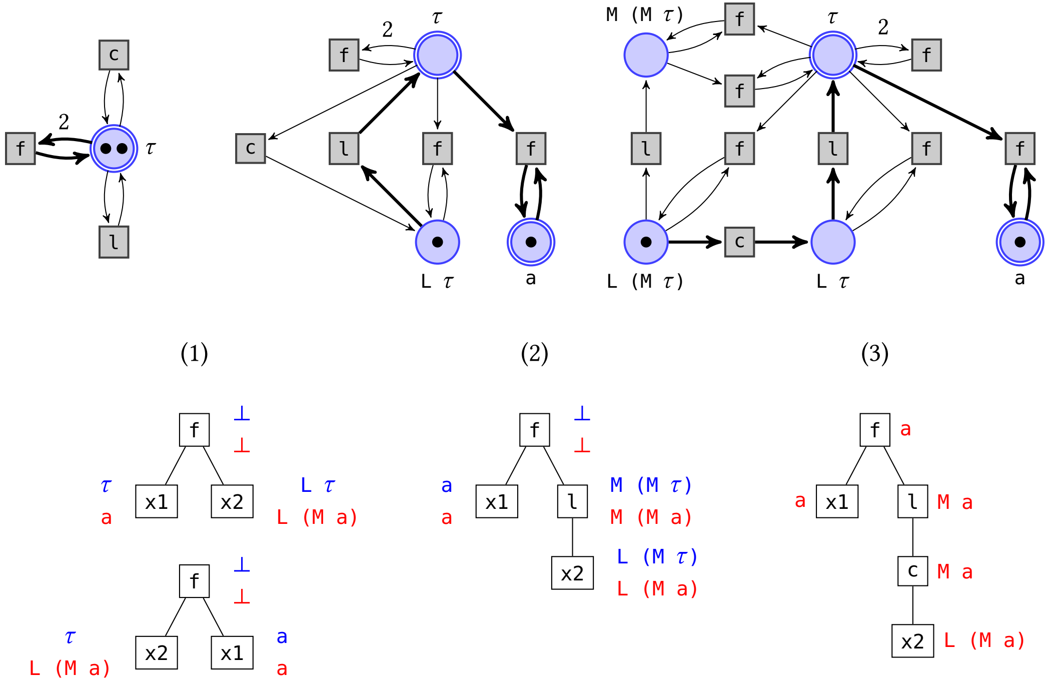

Fig. 3. Three iterations of abstraction refinement: ATNs (above) and corresponding solutions (below). Some irrelevant transitions are omitted from the ATNs for clarity. Solutions 1 and 2 are spurious, solution 3 is valid. Each solution is annotated with its concrete typing (in red); each spurious solution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

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. 3 (2) shows the ATN for our running example with places τ, `L` τ and `a`. In this ATN, the *rightmost* `f` transition takes `a` and τ as inputs and returns `a` as output. This transition represents the set of monomophic types $\{$`a` $\rightarrow t \rightarrow$ `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 $\{$`a` $\rightarrow$ `M a` $\rightarrow$ `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. 3.

Iteration 1. We start with the coarsest possible abstraction, where all types are abstracted to τ, yielding the ATN in Fig. 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. 3 (1). Since both candidate programs are ill-typed (as indicated by the annotation $\bot$ 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 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. 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. 3 (2).

Iteration 2. The new ATN in Fig. 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. 3 (2). To compute the proof of untypeability, we start as before, 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. 3 (3).

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

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 → 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?

```
-- | 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
```

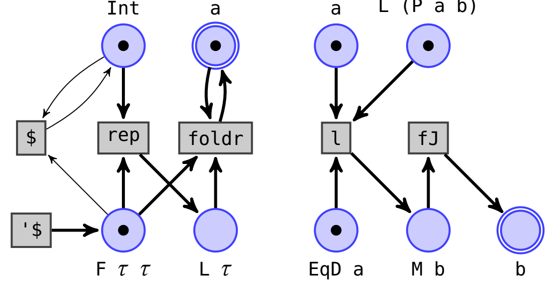

Fig. 4. (left) A library with higher-order functions and type-class constraints. (center) Fragment of an ATN for the query `(a → a) → a → Int → a`. (right) Fragment of an ATN for the query `Eq a => [(a,b)] → a → b`.

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 [Pierce 2004], which requires each variable to be used *at least once*, making programs like `\x1 x2 → 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 solutions like these, we use GHC's *demand analysis* [Sergey et al. 2017] 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. 6, demand analysis significantly improves the quality of solutions.

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. 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. 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 → ... → Bn → 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. 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 `'$`, which is actually taken by our solution, since `($)` 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. 4 both `a → a` and `(a → a) → a → 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 would have required an explicit lambda-subterm for a function argument, can now be written in a point-free style, only using variables, components and their applications.

2.5 Type Classes

Type classes are widely used in Haskell to support *ad-hoc* polymorphism [Wadler and Blott 1989]. For example, consider the type of component `lookup` in Fig. 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` $\alpha \to \alpha \to$ `[(`α`,`β`)]` $\to \beta$. 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 a lookup in a key-value list *assuming* the key is present, and poses a query `Eq a => [(a,b)] → a → b`. The intended solution to this query is `\xs k → 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. 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

[4]As we explain in Sec. 5.1, dictionaries can also be inhabited via instances and functional dependencies.

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

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

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

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

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

Fig. 5. λ_H: syntax and declarative type system.

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)] → Int → b` (where the key type is just an `Int`) H+ produces a flurry of irrelevant results, such as `\xs k → snd (xs !! k)`, which uses `k` as an *index* into the list, and not as a key as we intended.

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. 4.

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. 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* $\sigma = [\tau_1 \mapsto B_1, \ldots, \tau_n \mapsto B_n]$ 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).

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 $\bot$ that is strictly more specific than every other base type; we also consider a bottom substitution $\sigma_\bot$ and define $\sigma_\bot B = \bot$ 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 $\sigma_\bot$ 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 $\texttt{P}\ \alpha\ \texttt{B} \sqcap \texttt{P}\ \beta\ \texttt{A} = \bot$. 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 $\mathbf{B}_\bot = \mathbf{B} \cup \{\bot\}$ for the set of base types augmented with $\bot$. Note that $\langle \mathbf{B}_\bot, \sqsubseteq, \sqcup, \sqcap \rangle$ is a lattice with bottom element $\bot$ and top element τ and is isomorphic to Plotkin [1970]'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, intuitively, a component $\texttt{l :: }\forall\ \beta.\texttt{L}\ \beta \to \texttt{M}\ \beta$ maps `L A` to `M A`, `L (M A)` to `M (M A)`, and `A` to $\bot$. This gives rise to *type transformer* semantics for components, which is similar to predicate transformer semantics in predicate abstraction and SYNGAR [Wang et al. 2018], 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 $\mathsf{fresh}(\overline{\forall \tau}.T) \triangleq [\overline{\tau \mapsto \tau'}]T$, 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 $[\![c]\!]_\Lambda \colon \overline{\mathbf{B}_\bot} \to \mathbf{B}_\bot$ 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: $[\![\texttt{l}]\!](\texttt{L (M}\ \tau\texttt{)}) = \texttt{M (M}\ \tau\texttt{)}$, $[\![\texttt{l}]\!](\tau) = \texttt{M}\ \tau_1$ (where τ_1 is a fresh type variable), and $[\![\texttt{l}]\!](\texttt{A}) = \bot$ (because $\mathsf{mgu}(\texttt{L}\ \tau_2, \texttt{A}) = \sigma_\bot$). 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 $[\![\texttt{l}]\!](B)$.

Lemma 3.1 (Trans. Monotonicity). *If* $\overline{B_i^1 \sqsubseteq B_i^2}$ *then* $[\![c]\!](\overline{B_i^1}) \sqsubseteq [\![c]\!](\overline{B_i^2})$.

Lemma 3.2 (Trans. Soundness). *If* $\mathsf{fresh}(\Lambda(c)) = \overline{B_i} \to B$ *and* $\overline{\sigma B_i \sqsubseteq B'_i}$ *then* $\sigma B \sqsubseteq [\![c]\!](\overline{B'_i})$.

The proofs of these and following results can be found in the technical report [Guo et al. 2019].

Bidirectional Typing. We can use type transformers to define algorithmic type checking for λ_H, as shown in Fig. 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 [Pierce and Turner 2000], 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.

(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 $\boxed{\Lambda; \Gamma \vdash_{\mathcal{A}} E \Longleftarrow t}$

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

Fig. 6. Abstract type checking for λ_H. Treating $\alpha_{\mathcal{A}}$ as the identity function yields concrete type checking.

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 $\llbracket c \rrbracket = \text{fresh}(\Lambda(c))$). 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 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 3.3 (Type Checking is Sound and Complete). $\Lambda; \cdot \vdash E :: t$ *iff* $\Lambda; \cdot \vdash E \Longleftarrow t$.

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 $A_i \in \mathbf{B}_\bot$ that contains τ and $\bot$, 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`, $\mathcal{A}_0 = \{\tau, \bot\}$, $\mathcal{A}_1 = \{\tau, \mathtt{A}, \mathtt{L}\ \tau, \bot\}$, and $\mathcal{A}_2 = \{\tau, \mathtt{A}, \mathtt{L}\ \tau, \mathtt{L}\ (\mathtt{M}\ \tau), \mathtt{M}\ (\mathtt{M}\ \tau), \bot\}$ 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* $\alpha_{\mathcal{A}} : \mathbf{B}_\bot \to \mathbf{B}_\bot$ 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. 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 3.4 (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 $\mathbf{B}_\perp \preceq \mathcal{A}$ for any $\mathcal{A}$, the above Theorem 3.4 implies that abstract typing conservatively *over-approximates* concrete typing:

COROLLARY 3.5. *If $\Lambda; \cdot \vdash E \Longleftarrow t$ then $\Lambda; \cdot \vdash_{\mathcal{A}} E \Longleftarrow t$.*

4 SYNTHESIS

Next, we formalize the concrete and abstract synthesis problems, and use the notion of abstract type checking from Sec. 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 [Guo et al. 2019]).

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 Theorem 3.5 and Theorem 3.3, to show that any solution to a concrete synthesis problem is also a solution to any of its abstractions:

THEOREM 4.1. *If E is a solution to (Λ, t), then E is also a solution to $(\Lambda, t, \mathcal{A})$.*

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 follows:

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 $\alpha_{\mathcal{A}}\left(\llbracket c \rrbracket(A_1, \ldots, A_m)\right) \equiv A$; 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: $\exists f \in F. M_F(f) = 1 \land \forall p \in P. p \neq f \Rightarrow M_F(p) = 0$. 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.

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

Theorem 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 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. 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 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 4.3).

ATN versus TTN. Our ATN construction is inspired by but different from the TTN construction in SyPet [Feng et al. 2017]. 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 $\alpha_{\mathcal{A}}\left(\llbracket c \rrbracket(A_1, \ldots, A_m)\right)$. This result is either $\bot$, 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

[5] Sec. 5.3 details our encoding of ATN reachability into constraints.

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

```
1: function SynAbstract(Λ, t, 𝒜)
2:     𝒩 ← 𝒩(Λ, t, 𝒜)
3:     π ← ShortestValidPath(𝒩)
4:     if π = ⊥ then
5:         return ⊥
6:     else
7:         for E ∈ terms(π) do
8:             if Λ; · ⊢_𝒜 E ⟸ t then
9:                 return E
```

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

```
1: function Synthesize(Λ, t, 𝒜₀)
2:     𝒜 ← 𝒜₀
3:     while true do
4:         E ← SynAbstract(Λ, t, 𝒜)
5:         if E = ⊥ then
6:             return ⊥
7:         else if Λ; · ⊢ E ⟸ t then
8:             return E
9:         else
10:            𝒜 ← Refine(𝒜, E, t)
```

Fig. 7. (left) Algorithm for the abstract synthesis problem. (right) The TYGAR algorithm.

can be used zero or more times. Hence we allow the ATN to duplicate tokens in the initial marking I 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. 5.3); a linear type system can be supported by eliminating both.

4.2 The TYGAR Algorithm

The abstract synthesis algorithm from Fig. 7 either returns $\bot$, 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 $\mathcal{A}$ (starting with some $\mathcal{A}_0$) until it is specific enough that a solution to an abstract synthesis problem is also well-typed in the concrete type system.

Fig. 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 $\bot$ 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 4.1), so the algorithm returns $\bot$. 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. 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 many 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. 6) we found that synthesis is more efficient if we pick the initial cover $\mathcal{A}_Q(\overline{b_i} \to b) = \text{close}(\{\tau, \overline{b_i}, b, \bot\})$[6], which represents the query type $t = \overline{b_i} \to b$ concretely. Intuitively, the reason is that the distinctions

[6]Here 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: $\quad \Lambda \leftarrow \Lambda \cup (r :: b \to b)$
3: $\quad e^* \leftarrow r\ e_{body}$
4: $\quad$ **for** $e_j \in$ subterms(e^*) **do**
5: $\quad\quad \Lambda; \overline{x_i : b_i} \vdash e_j \Longrightarrow U[e_j]$
6: $\quad U \leftarrow$ Generalize(U, e^*)
7: $\quad$ **return** close($\mathcal{A} \cup$ range(U))

Input: U, e s.t. $\mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_3$
Output: U' s.t. $\mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_3$

1: **function** Generalize(U, e)
2: $\quad$ **if** $e = x$ **then**
3: $\quad\quad$ **return** U
4: $\quad$ **else if** $e = c\ \overline{e_j}$ **then**
5: $\quad\quad \overline{B_j} \leftarrow$ weaken $\overline{U[e_j]}$ while $[\![c]\!](\overline{B_j}) \sqsubseteq U[e]$
6: $\quad\quad U' \leftarrow U[\overline{e_j \mapsto B_j}]$
7: $\quad\quad$ **for** e_j **do** Generalize(U', e_j)

Fig. 8. Refinement algorithm.

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 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 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 E_0, since E_0 is abstractly well-typed at any $\mathcal{A}$ by Theorem 3.5, and SynAbstract always returns a shortest abstractly well-typed program, when one exists by Theorem 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 then guarantees termination.

4.3 Refining the Abstract Cover

This section details the refinement step of the TYGAR algorithm. The pseudocode is given in Fig. 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 $U: \mathbf{e} \to \mathbf{B}_\bot$ from subterms of E to types, such that if 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

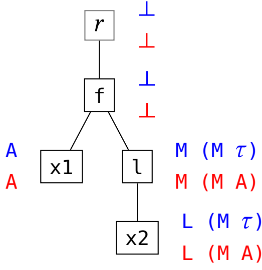

Fig. 9. Refine in the second iteration of the running example.

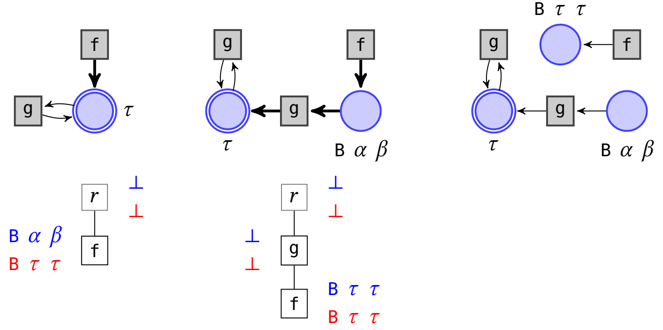

Fig. 10. Synthesize on an unsatisfiable problem.

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:

$\mathcal{I}_1$: *(U subsumes concrete typing)* For any $e \in \text{subterms}(e^*)$, if $\Lambda; \Gamma \vdash e \Longrightarrow B$, then $B \sqsubseteq U[e]$;
$\mathcal{I}_2$: *(U abstracts type transformers)* For any application subterm $e = c\ \overline{e_j}$, $[\![c]\!](\overline{U[e_j]}) \sqsubseteq U[e]$;
$\mathcal{I}_3$: *(U proves untypeability)* $U[e^*] = \bot$.

Lemma 4.6. *If $\mathcal{I}_1 \wedge \mathcal{I}_2 \wedge \mathcal{I}_3$ then U is a proof of untypeability: if* $\text{range}(U) \subseteq \mathcal{A}'$ *then* $\Lambda; \cdot \nvdash_{\mathcal{A}'} E \Longleftarrow t$.

Proof Sketch. We can show by induction on the derivation that for any $\mathcal{A}' \supseteq \text{range}(U)$ and node e, $\Lambda; \Gamma \vdash_{\mathcal{A}'} e \Longrightarrow B \sqsubseteq U[e]$ (base case follows from $\mathcal{I}_1$, and inductive case follows from $\mathcal{I}_2$). Hence, $\Lambda; \Gamma \vdash_{\mathcal{A}'} e^* \Longrightarrow B \sqsubseteq U[e^*] = \bot$ (by $\mathcal{I}_3$), so $\Lambda; \Gamma \vdash_{\mathcal{A}'} e_{body} \Longrightarrow B \not\sqsubseteq b$, and $\Lambda; \cdot \nvdash_{\mathcal{A}'} E \Longleftarrow t$. □

Correctness of Generalize. Now that we know that invariants $\mathcal{I}_1$–$\mathcal{I}_3$ 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 $\mathcal{I}_2$. 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 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 $\mathcal{I}_1$–$\mathcal{I}_3$. $\mathcal{I}_1$ is maintained by property (1) of weakening (we start from concrete types and only move up in the lattice). $\mathcal{I}_2$ 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, $\mathcal{I}_3$ 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. 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 $\mathcal{A} = \{\tau, \texttt{A}, \texttt{L}\ \tau, \bot\}$, $E = \lambda x_1\ x_2.\texttt{f}\ x_1\ (\texttt{l}\ x_2)$ 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. 9 (left). It then initializes the mapping U with concrete inferred types, depicted as red labels; as expected $U[e^*] = \bot$. 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 $[\![\texttt{f}]\!](\texttt{A}, \texttt{M (M}\ \tau)) = \bot$), 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 $\texttt{L (M}\ \tau)$ but no further, since $[\![\texttt{l}]\!](\texttt{L (M}\ \tau)) = \texttt{M (M}\ \tau)$ but $[\![\texttt{l}]\!](\texttt{L}\ \tau) = \texttt{M}\ \tau \not\sqsubseteq \texttt{M (M}\ \tau)$.
- 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. 10. SynAbstract(Λ, Z, $\mathcal{A}_0$) returns a program f, which is spurious, hence we invoke Refine($\mathcal{A}_0$, f, Z). The results of concrete type inference are shown as red labels in Fig. 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(Λ, 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: $\Lambda; \cdot \vdash \texttt{f} \Longrightarrow \texttt{B}\ \tau\ \tau$. (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 $\texttt{B}\ \tau\ \tau$); we could not, for example, replace type variables with a single construct ?, as in gradual typing [Siek and Taha 2006].

5 IMPLEMENTATION

We have implemented the TYGAR synthesis algorithm in Haskell, in a tool called H+. The tool relies on the Z3 SMT solver [de Moura and Bjørner 2008] 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.

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. 2.4 and Sec. 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` α β (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` $\tau \to T_i$ in Λ. For example, a type class declaration `class Eq` α `where (==) :: a → a → Bool` creates a fresh type constructor `EqD` α and a component `(==) :: EqD` $\alpha \to \alpha \to \alpha \to$ `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 $\Lambda(c) = \text{desugar}\left(\tilde{\Lambda}(c)\right)$, where the translation function desugar, which eliminates type class constraints and higher-order types, is defined as follows:

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

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

$$\text{base}(T_1 \to T_2) = \mathtt{F}\ \text{base}(T_1)\ \text{base}(T_2) \qquad \text{base}(B) = B$$

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 $\Lambda(c') = \text{base}(\Lambda(c))$. For example: `any' :: F (F α Bool) (F [α] Bool)`.
(6) Finally, the λ_H query type t is defined as 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 paper. Secondly, in theory our encoding of higher-order functions (Sec. 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 [Barendregt 1985] including an *apply* component `($) :: F` α $\beta \to \alpha \to \beta$ 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.

5.2 ATN Construction

Incremental updates. Sec. 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 parents be the direct successors of A_{new} in the $\sqsubseteq$ partial order; for example, in the cover $\{\tau, \mathtt{P}\ \alpha\ \beta, \mathtt{P\ A}\ \beta, \mathtt{P}\ \alpha\ \mathtt{B}, \mathtt{P\ A\ B}, \bot\}$, the parents of `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 parents. Similarly, all

new transitions can be derived from those that take one of the types in 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 $A \in$ 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 $A_i \in$ parents; consider $\overline{A'_i}$ obtained from $\overline{A_i}$ by substituting at least one $A_i \in$ 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. 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. 3 (1), both c and l have the type $\tau \rightarrow \tau$. Our implementation *coalesces* equivalent transitions: an optimization known in the literature as *observational equivalence reduction* [Alur et al. 2017; Wang et al. 2018]. 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.

5.3 SMT Encoding of ATN Reachability

Our encoding differs slightly from that in previous work on SyPet. Most notably, we use an SMT (as opposed to SAT) encoding, in particular, representing transition firings as integer 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 $\text{pre}(x) = \{y \in P \cup T \mid E(y, x) > 0\}$ and the *post-image* of x be $\text{post}(x) = \{y \in P \cup T \mid E(x, y) > 0\}$.

The formula ϕ is a conjunction of the following constraints:

(1) At each time step, a valid transition is fired: $\bigwedge_{k=0}^{\ell-1} 1 \leq \texttt{fire}_k \leq |T|$
(2) If a transition t is fired at time step k then all places $p \in \text{pre}(t)$ have sufficiently many tokens: $\bigwedge_{k=0}^{\ell-1} \bigwedge_{t=1}^{|T|} \texttt{fire}_k = t \implies \bigwedge_{p \in \text{pre}(t)} \texttt{tok}_k^p \geq E(p, t)$
(3) If a transition t is fired at time step k then all places $p \in \text{pre}(t) \cup \textit{post}(t)$ will have their markings updated at time step $k+1$: $\bigwedge_{k=0}^{\ell-1} \bigwedge_{t=1}^{|T|} \texttt{fire}_k = t \implies \bigwedge_{p \in \text{pre}(t) \cup \text{post}(t)} \texttt{tok}_{k+1}^p = \texttt{tok}_k^p - E(p, t) + E(t, p)$
(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: $\bigwedge_{k=0}^{\ell-1} \bigwedge_{p \in P} (\bigwedge_{t \in \text{pre}(p) \cup \text{post}(p)} \texttt{fire}_k \neq t) \implies \texttt{tok}_{k+1}^p = \texttt{tok}_k^p$

(5) The initial marking is I: $\bigwedge_{p \in P} \texttt{tok}_0^p = I(p)$.
(6) The final marking is valid: $\bigvee_{f \in F} \left(\texttt{tok}_\ell^f = 1 \wedge \bigwedge_{p \in P \setminus \{f\}} \texttt{tok}_\ell^p = 0 \right)$.

Optimizations. Although the validity of the final marking can be encoded as in (6) above, we found that quality of solutions improves if instead we iterate through $f \in F$ in the order from *most to least precise*; in each iteration we enforce $\texttt{tok}_\ell^f = 1$ (and $\texttt{tok}_\ell^p = 0$ 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 correspond to concretely well-typed programs.

As we mentioned in Sec. 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: $\bigwedge_{p \in \{P | I(p) > 0\}} \texttt{tok}_0^p \geq I(p)$ and $\bigwedge_{p \in \{P | I(p) = 0\}} \texttt{tok}_0^p = 0$. 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.

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` $\rightarrow$ `Parser a` $\rightarrow$) or unrealizable (*e.g.* `a` $\rightarrow$ `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.
(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. 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.

[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`, `&`, `(.)`.

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

Fig. 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.

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. 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. 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 $\mathcal{A}_\top = \{\tau, \bot\}$.

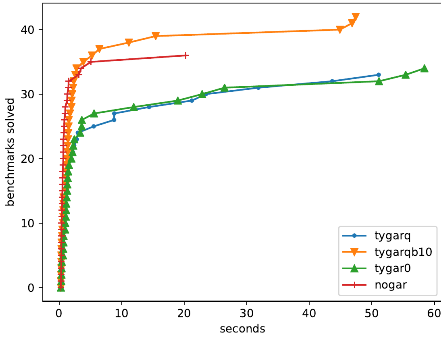

Fig. 12. Queries solved over time for our initial variants and the best refinement bound.

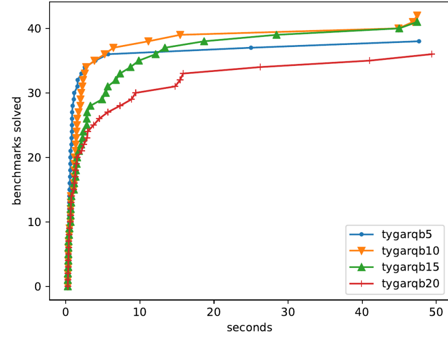

Fig. 13. Queries solved over time for varying refinement bounds. The variant's number indicates the refinement bound on the abstract cover.

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

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 [Guo et al. 2019]. 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. 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. 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 throughout Sec. 2:

- `a → [Maybe a] → a` corresponds to benchmark 19 (fromFirstMaybes); the solution from Sec. 2 is generated at rank 18.
- `(a → a) → a → Int → a` corresponds to benchmark 11 (applyNTimes); the solution from Sec. 2 is generated at rank 10.
- `Eq a => [(a,b)] → a → b` corresponds to benchmark 18 (lookup); the solution from Sec. 2 is generated at rank 1.

H+ has also produced code snippets that surprised us: for example, on the query `(a → b, a) → b`, 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,

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

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.

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 [Gulwani 2011; Katayama 2012; Lee et al. 2018; Osera and Zdancewic 2015], logical specifications [Galenson et al. 2014; Srivastava et al. 2010] or program sketches [Solar-Lezama 2008].

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 [Mitchell 2004] that search for type isomorphisms [Di Cosmo 1993] 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 [Raychev et al. 2014]. 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 [Murali et al. 2017]. 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 [Urzyczyn 1997]. 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 [Augusstson 2005] which implements a decision procedure for intuitionistic propositional calculus [Dyckhoff and Pinto 1998] 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 [Heineman et al. 2016]. 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 [Polikarpova et al. 2016] and Myth2 [Frankle et al. 2016] use different flavors of refinement types to synthesize recursive functions, while Agda [Norell 2008] 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 [Perelman et al. 2012] 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 [Gvero et al. 2013] 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 [Mandelin et al. 2005] 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 [Feng et al. 2017], 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 [Clarke et al. 2010], there are two lines of work in particular closely related to ours. First, [Ganty et al. 2007; Kloos et al. 2013] describe an iterative abstraction-refinement process for verifying Petri nets, using SMT [Esparza et al. 2014]. 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 [Wang et al. 2018] 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 [Cousot 1997]. 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, [Garcia et al. 2016] 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. 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).

ACKNOWLEDGMENTS

The authors would like to thank Neil Mitchell for providing the Hoogle data and helpful feedback on the H+ web interface. We are also grateful to the anonymous reviewers of this and older versions of the paper for their careful reading and many constructive suggestions. This research was supported by NSF grants 1814358 and 1911149.

REFERENCES

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. 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.* 319–336. https://doi.org/10.1007/978-3-662-54577-5_18

Lennart Augusstson. 2005. Djinn. https://github.com/augustss/djinn.

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

Edmund M. Clarke, Robert P. Kurshan, and Helmut Veith. 2010. The Localization Reduction and Counterexample-Guided Abstraction Refinement. In *Time for Verification, Essays in Memory of Amir Pnueli*. 61–71. https://doi.org/10.1007/978-3-642-13754-9_4

Patrick Cousot. 1997. Types As Abstract Interpretations. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '97)*. ACM, New York, NY, USA, 316–331. https://doi.org/10.1145/263699.263744

Patrick Cousot and Radhia Cousot. 1977. 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)*. ACM, New York, NY, USA, 238–252. https://doi.org/10.1145/512950.512973

Leonardo Mendonça de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *TACAS (LNCS)*, Vol. 4963. Springer, 337–340.

Roberto Di Cosmo. 1993. Deciding Type isomorphisms in a type assignment framework. *Journal of Functional Programming* 3, 3 (1993), 485–525. https://doi.org/10.1017/S0956796800000861 Special Issue on ML.

Roy Dyckhoff and Luís Pinto. 1998. Proof Search in Constructive Logics. In *In Sets and proofs*. Cambridge University Press, 53–65.

Javier Esparza, Ruslán Ledesma-Garza, Rupak Majumdar, Philipp J. Meyer, and Filip Niksic. 2014. 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*. 603–619. https://doi.org/10.1007/978-3-319-08867-9_40

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

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. 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*. 802–815. https://doi.org/10.1145/2837614.2837629

Joel Galenson, Philip Reames, Rastislav Bodik, Björn Hartmann, and Koushik Sen. 2014. CodeHint: Dynamic and Interactive Synthesis of Code Snippets. In *Proceedings of the 36th International Conference on Software Engineering (ICSE 2014)*. ACM, New York, NY, USA, 653–663. https://doi.org/10.1145/2568225.2568250

Pierre Ganty, Jean-François Raskin, and Laurent Van Begin. 2007. 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*. 124–143. https://doi.org/10.1007/978-3-540-73094-1_10

Ronald Garcia, Alison M. Clark, and Éric Tanter. 2016. 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*. 429–442. https://doi.org/10.1145/2837614.2837670

Susanne Graf and Hassen Saidi. 1997. Construction of abstract state graphs with PVS. In *Computer Aided Verification*. 72–83.

Sumit Gulwani. 2011. 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*. 317–330. https://doi.org/10.1145/1926385.1926423

Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. arXiv:1911.04091

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

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

Susumu Katayama. 2012. 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*. 43–52. https://doi.org/10.1145/2103746.2103758

Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac. 2013. Incremental, Inductive Coverability. In *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*. 158–173. https://doi.org/10.1007/978-3-642-39799-8_10

Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-based Program Synthesis Using Learned Probabilistic Models. In *PLDI*.

David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *PLDI*.

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

Vijayaraghavan Murali, Swarat Chaudhuri, and Chris Jermaine. 2017. 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*. 151–162. https://doi.org/10.1145/3106237.3106284

Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*. 230–266. https://doi.org/10.1007/978-3-642-04652-0_5

Peter-Michael Osera and Steve Zdancewic. 2015. 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*. 619–630. https://doi.org/10.1145/2737924.2738007

Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. 2012. Type-directed completion of partial expressions. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. 275–286. https://doi.org/10.1145/2254064.2254098

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

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

Gordon Plotkin. 1970. *Lattice Theoretic Properties of Subsumption*. Edinburgh University, Department of Machine Intelligence and Perception. https://books.google.com/books?id=2p09cgAACAAJ

Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. 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*. 522–538. https://doi.org/10.1145/2908080.2908093

Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code Completion with Statistical Language Models. *SIGPLAN Not.* 49, 6 (June 2014), 419–428. https://doi.org/10.1145/2666356.2594321

Ilya Sergey, Dimitrios Vytiniotis, Simon L. Peyton Jones, and Joachim Breitner. 2017. Modular, higher order cardinality analysis in theory and practice. *J. Funct. Program.* 27 (2017), e11. https://doi.org/10.1017/S0956796817000016

Kensen Shi, Jacob Steinhardt, and Percy Liang. 2019. FrAngel: Component-based Synthesis with Control Structures. *Proc. ACM Program. Lang.* 3, POPL, Article 73 (Jan. 2019), 29 pages. https://doi.org/10.1145/3290386

Jeremy G. Siek and Walid Taha. 2006. Gradual Typing for Functional Languages. In *IN SCHEME AND FUNCTIONAL PROGRAMMING WORKSHOP*. 81–92.

Armando Solar-Lezama. 2008. *Program Synthesis by Sketching*. Ph.D. Dissertation. Berkeley, CA, USA. Advisor(s) Bodik, Rastislav. AAI3353225.

Saurabh Srivastava, Sumit Gulwani, and Jeffrey S. Foster. 2010. 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*. 313–326. https://doi.org/10.1145/1706299.1706337

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

Philip Wadler and Stephen Blott. 1989. 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*. 60–76. https://doi.org/10.1145/75277.75283

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