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 languages like Java do scale, but only apply to components over monomorphic data and functions: polymorphic data and functions 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. We have evaluated H+ on a set of 44 queries using a set of popular Haskell libraries with a total of 291 components. Our results demonstrate that H+ returns an interesting solution within the first five results for 32 out of 44 queries. Moreover, TYGAR allows H+ to rapidly return well-typed terms, with the median time to first solution of just 1.4 seconds.

1 INTRODUCTION

Consider the task of implementing a function \texttt{firstJust def mbs}, which extracts the first non-empty value from a list of options \texttt{mbs}, and if none exists, returns a default value \texttt{def}. Rather than writing a recursive function, you suspect you can implement it more concisely and idiomatically using standard library functions. 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 \texttt{firstJust}, i.e. \texttt{a \to [Maybe a] \to a}. The search results will be disappointing, however, since no single API function matches this type. In fact, to implement \texttt{firstJust} you need a snippet that composes three library functions from the standard \texttt{Data.Maybe} library, like so: \texttt{\def mbs \to 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 \textit{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.
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; Mandelin et al. 2005].

**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 $a \rightarrow b \rightarrow (a,b)$ 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 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 that take functions as input by representing functions (arrows) as a binary type constructor. Similarly, TYGAR can handle constrained components that arise due to Haskell’s ubiquitous type-classes, by following the dictionary-passing translation, which again, relies crucially on support for parametric polymorphism. Thus, 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 any 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 synthesizing relevant programs by forcing the synthesizer to generate terms where all the input variables syntactically appear, and then using GHC’s demand analysis [Sergey et al. 2017] eliminate terms where some of the inputs are unused.

We have evaluated H+ on a set of 44 queries collected from different sources (including 24 real queries from Hoogle), 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 out of 44 queries, the top five results contains a useful solution. Further, our evaluation demonstrates that TYGAR is crucial for efficient synthesis: a naive approach that 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.

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

---

1Unfortunately, ground truth solutions are not available for Hoogle benchmarks; we judge usefulness by manual inspection.
--- | Value stored in the option or default if the option is empty

fromMaybe :: \(\alpha\) \(\rightarrow\) \(\text{Maybe} \ \alpha\) \(\rightarrow\) \(\alpha\)

--- | All values from a list of options

\[\text{catMaybes} :: \text{List} (\text{Maybe} \ \alpha) \rightarrow \text{List} \ \alpha\]

--- | Head of the list or empty option if the list is empty

\[\text{listToMaybe} :: \text{List} \ \alpha \rightarrow \text{Maybe} \ \alpha\]

--- | Value stored in the option

\[\text{fromMaybe} :: \alpha \rightarrow \text{Maybe} \ \alpha \rightarrow \alpha\]

--- | All values from a list of options

\[\text{catMaybes} :: \text{List} (\text{Maybe} \ \alpha) \rightarrow \text{List} \ \alpha\]

--- | Head of the list or empty option if the list is empty

\[\text{listToMaybe} :: \text{List} \ \alpha \rightarrow \text{Maybe} \ \alpha\]

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

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, \(a, \text{Maybe} \ a, \text{Maybe} (\text{Maybe} \ a), [a], \text{and} [\text{Maybe} \ a]\). For the rest of this section, we abbreviate the names of components and type constructors to their first letter (for example, we will write \(\text{L} (\text{M} \ a)\) for \([\text{Maybe} \ a]\)) and refer to the query arguments as \(x_1, x_2\).

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 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 \(\text{L} (\text{M} \ a)\) and producing one in \(\text{L} \ a\); however transition \(\text{f} <\alpha>\) cannot fire as there is no token in \(\text{M} \ a\).

Synthesis via Petri-Net Reachability. Given a synthesis query \(T_1 \rightarrow \ldots \rightarrow T_n \rightarrow T\), 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 \(f \ x_1 (l \ (c \ x_2))\). 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 α → α backwards from α yields L α, L (L α), 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 t and l components, each giving rise to two transitions, by instantiating their type variable α with two different TTN places, α and Maybe α. 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 (α and L (M α)), as this will preclude the synthesis of terms that generate intermediate values of some other type, e.g. L α 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, 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 \{L t | t ∈ Type\}. This representation supports different levels of detail: for example, the type L (M a) can be abstracted into itself, L (M τ), L τ, or τ.

Abstract Transition Nets. A Petri net constructed out of abstract types, which we dub an abstract transition net (ATN), can finitely represent all types in our universe, and hence all possible solutions to the synthesis problem. The ATN construction is grounded in the theory of abstract interpretation and ensures that the net soundly over-approximates the concrete type system, i.e. that every well-typed program corresponds to some valid path through the ATN. Fig. 3 (2) shows the ATN for our running example with places τ, L τ and α. In this ATN, the rightmost \( f \) transition takes α and τ as inputs and returns α as output. This transition represents the set of monomorphic types \{α → t → α | t ∈ Type\} and over-approximates the set of instances of \( f \) where the first argument unifies with α and the second argument unifies with τ (which in this case is a singleton set \{α → M a → 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 \( f \times l \times 2 \times l \times 2 \), which is ill-typed since the arguments to \( f \) have the types α 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. \( \text{H}^+ \) 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: \( f \times 1 \times 2 \) and \( f \times 2 \times 1 \). Next, we type-check these programs to determine whether they
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.

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 \( f \), \( M \ a \). 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 \( M \ a \). 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 \( \tau \) and \( L \ \tau \) (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 $M \ (M \ \tau)$. 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 $M \ (M \ \tau)$. The generalization process stops at the the leaves of the AST (or alternatively when the type of some node cannot be generalized). Adding the types $M \ (M \ \tau)$ and $L \ (M \ \tau)$ 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 $\ x1 \ x2 \ \rightarrow \ x1$ has the right type, but it is clearly uninteresting. Hence, the important challenge for H+ is: how do we filter out uninteresting solutions without requiring additional input from the user?

Relevant Typing. SyPet offers an interesting approach to this problem: they observe that a programmer is unlikely to include an argument in a query if this argument is not required for the solution. To leverage this observation, they propose to use a relevant type system [Pierce 2004], which requires each variable to be used at least once, making programs like $\ x1 \ x2 \ \rightarrow \ x1$ ill-typed. TTNs naturally enforce relevancy during search: in fact, TTN reachability as described so far encodes a stricter linear type system, where all arguments must be used exactly once. To relax this requirement, we allow the initial marking to contain an arbitrary positive number of tokens for each argument type.

Demand Analysis. Unfortunately, with expressive polymorphic components the synthesizer discovers ingenious ways to circumvent the relevancy requirement. For example, the terms $\text{fst} \ (x1, x2)$, $\text{const} \ x1 \ x2$, and $\text{fromLeft} \ x1 \ (\text{Right} \ x2)$ 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 \((a \to a) \to a \to \text{Int} \to a\), 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 \(\lambda x \, n \to \text{foldr} \ (\$) \ x \ (\text{replicate} \ n \ g)\).

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 Nullar 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 \(F \alpha \beta\) to represent arrow types as if they were base types; and (2) for each component \(c :: B_1 \to \ldots \to B_n \to B\) in the original library, we add a nullary component \(\text{'}c :: F \ B_1 \ (\ldots \ F \ B_n \ B)\). 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 \(\text{'}\$, which is actually taken by our solution, since \((\$\) is used as an argument to \text{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 \(a \to a\) and \((a \to a) \to a \to a\) are abstracted into the same place \(F \tau \tau\).

**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.

---

Fig. 4. (left) A library with higher-order functions and type-class constraints. (center) Fragment of an ATN for the query \((a \to a) \to a \to \text{Int} \to a\). (right) Fragment of an ATN for the query \(\text{Eq} a \Rightarrow [(a,b)] \to a \to b\).
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 \( \alpha \) and a list of key-value pairs of type \( \{ (\alpha, \beta) \} \), 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 \( \text{Eq} \) on the type of keys, enforcing that any concrete key type be an instance of the type class \( \text{Eq} \) 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, \( \mathbf{H}^+ \) can use dictionary passing to desugar synthesis with type classes into a synthesis problem supported by ATNs. For example, the type of \( \text{lookup} \) is desugared into an unbounded type with an extra argument: \( \text{EqD} \alpha \to \alpha \to \{ (\alpha, \beta) \} \to \beta \). Here \( \text{EqD} \alpha \) is a dictionary: a record datatype that stores the implementation of equality on \( \alpha \); the exact definition of this datatype is unimportant, we only care whether \( \text{EqD} \alpha \) for a given \( \alpha \) 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 \( \text{Eq} \  a \Rightarrow \{ (a,b) \} \to a \to b \). The intended solution to this query is \( \{xs\  k \to \text{fromJust} \ (\text{lookup} k \  xs)\} \), 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 \( \text{lookup} \) with \( a \to a, \beta \to b \) — has \( \text{EqD} \  a \) 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 \( \text{Eq} \), or in other words, that \( \text{EqD} \  a \) 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 \( \text{EqD} \  a \).² A welcome side-effect of relevant typing is that any solution must use the token in \( \text{EqD} \  a \), which matches our intuition that the user would not specify the bound \( \text{Eq} \  a \) if they did not mean to compare keys for equality. This example illustrates that the combination of (bounded) polymorphism and relevant typing gives users a surprisingly powerful mechanism to disambiguate their intent. Given the query above (and a library of 291 components), \( \mathbf{H}^+ \) returns the intended solution as the first result. In contrast, given a monomorphic variant of this query \( \{ (\text{Int}, \ b) \} \to \text{Int} \to b \) (where the key type is just an \( \text{Int} \) ) \( \mathbf{H}^+ \) produces a flurry of irrelevant results, such as \( \{xs\  k \to \text{snd} \ (xs \  !! \ k)\} \), 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 \( \lambda_H \) and its type system, and use the framework of abstract interpretation to define an algorithmic abstract type system for \( \lambda_H \). 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 \( \lambda_H \) Language

\( \lambda_H \) 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 \( \tau \), as well as applications of a type constructor to zero or more base types \( C \  \overline{B} \). We write \( X \) 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

²As we explain in Sec. 5.1, dictionaries can also be inhabited via instances and functional dependencies.
categories $b$ and $t$ are the ground counterparts to $B$ and $T$ (i.e. they contain no type variables). A component library $\Lambda$ is a finite map from a set of components $c$ to the components’ poly-types. A typing environment $\Gamma$ 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 $\tau_i$ to $B_i$ and is identity elsewhere. We write $\sigma T$ to denote the application of $\sigma$ 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 analogically picks ground base types to substitute for all the universally-quantified type variables in the component signature (the rule implicitly requires that $\sigma T$ 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 $\sigma$ such that $T' = \sigma T$. The relation $\sqsubseteq$ is a partial order on types. For example, in a library with two nullary type constructors $\text{A}$ and $\text{B}$, and a binary type constructor $\text{P}$, we have $\text{P} \text{A} \text{B} \sqsubseteq \text{P} \alpha \text{B} \sqsubseteq \text{P} \alpha \beta \sqsubseteq \tau$. This partial order induces an equivalence relation $T_1 \equiv T_2 \triangleq T_1 \sqsubseteq T_2 \land T_2 \sqsubseteq T_1$ (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 $\sigma$ 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 $\text{mgu}(B_1, B_2)$. We write $\text{mgu}(B_1, B_2)$ for the MGU of a sequence of type pairs, where the MGU of an empty sequence is the identity substitution (mgu(·) = []). The meet of two base types is defined as $B_1 \sqcap B_2 = \sigma B_1 = \sigma B_2$, where $\sigma = \text{mgu}(B_1, B_2)$. For example, $\text{P} \alpha \text{B} \sqcap \text{P} \alpha \beta = \text{P} \alpha \text{B}$ while $\text{P} \alpha \beta \sqcap \text{P} \beta \text{A} = \bot$. 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 $B_\bot = B \cup \{ \bot \}$ for the set of base types augmented with $\bot$. Note that $\langle B_\bot, \sqsubseteq, \bot, \sqcap, \sqcup, \sqcap, \sqcup \rangle$ is a lattice with bottom element $\bot$ and top element $\tau$ 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 $\bot : \forall \beta, \text{L} \beta \rightarrow \text{M} \beta$ maps $\text{L} \text{A}$ to $\text{M} \text{A}$, $\text{L} (\text{M} \text{A})$ to $\text{M} (\text{M} \text{A})$, and $\text{A}$ to $\bot$. This gives rise to type transformer semantics for components, which is similar to predicate transformer semantics in predicate abstraction and
(Abstract) Type Inference \[ \Lambda ; \Gamma \vdash \alpha \mathcal{A} e \implies B \]

- **I-VAR**
  \[ \Gamma(x) = b \]
  \[ \Lambda ; \Gamma \vdash \alpha \mathcal{A} x \implies \alpha \mathcal{A}(b) \]

- **I-APP**
  \[ \Lambda ; \Gamma \vdash \alpha \mathcal{A} e_1 \implies B_i \]
  \[ \Lambda ; \Gamma \vdash \alpha \mathcal{A} c \overline{e}_i \implies \alpha \mathcal{A} \left[ \overline{c} \right](\overline{B_i}) \]

(Abstract) Type Checking \[ \Lambda ; \Gamma \vdash \alpha \mathcal{A} E \leftarrow t \]

- **C-FUN**
  \[ \Lambda ; \Gamma , x : b \vdash \alpha \mathcal{A} E \leftarrow t \]
  \[ \Lambda ; \Gamma \vdash \alpha \mathcal{A} \lambda x.E \leftarrow b \rightarrow t \]

- **C-BASE**
  \[ \Lambda ; \Gamma \vdash \alpha \mathcal{A} e \implies B \]
  \[ b \subseteq B \]
  \[ \Lambda ; \Gamma \vdash \alpha \mathcal{A} e \leftarrow b \]

Fig. 6. Abstract type checking for \( \lambda_H \). Treating \( \alpha \mathcal{A} \) as the identity function yields concrete type checking.

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 \( \text{fresh}(\overline{\tau}.T) \triangleq [\overline{\tau} \mapsto \overline{\tau'}]T \), where \( \overline{\tau'} \) are fresh type variables. Let \( c \) be a component and \( \text{fresh}(\Lambda(c)) = \overline{B_i} \rightarrow \overline{B'} \); then a type transformer for \( c \) is a function \( \left[ c \right]_\Lambda : \overline{B_{\perp}} \rightarrow \overline{B_{\perp}} \) defined as follows:

\[
\left[ c \right]_\Lambda(\overline{B_i}) = \sigma \overline{B'} \quad \text{where} \quad \sigma = \text{mgu}(\overline{B_1}, \overline{B'_1})
\]

We omit the subscript \( \Lambda \) where the library is clear from the context. For example, for the component \( \top \) above:

\[
\left[ \top \right]_{\text{L}}(\overline{M \top}) = \overline{M \top}, \quad \left[ \top \right]_{\text{L}}(\overline{\tau_1}) = \overline{\tau_1} \text{ (where } \overline{\tau_1} \text{ is a fresh type variable), and } \left[ \top \right]_{\text{L}}(\overline{\Lambda}) = \bot \text{ (because } \text{mgu}(\overline{\tau_2}, \overline{\Lambda}) = \overline{\bot}).\]

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 \( \top \) is more specific than some \( B \), its result is guaranteed to be more specific than \( \left[ \top \right](B) \).

**Lemma 3.1 (Trans. Monotonicity).** If \( \overline{B_1} \subseteq \overline{B'_1} \) then \( \left[ c \right](\overline{B_1}) \subseteq \left[ c \right](\overline{B'_1}) \).

**Lemma 3.2 (Trans. Soundness).** If \text{fresh}(\Lambda(c)) = \overline{B_i} \rightarrow B \text{ and } \sigma \overline{B_1} \subseteq \overline{B'_1} \text{ then } \sigma B \subseteq \left[ c \right](\overline{B'_1}) \).

The proofs of these and following results can be found in Appendix A.

**Bidirectional Typing.** We can use type transformers to define algorithmic type checking for \( \lambda_H \), 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 \( \Lambda; \Gamma \vdash e \implies B \) generates the (base) type \( B \) from the term \( e \), while the checking judgment \( \Lambda; \Gamma \vdash E \leftarrow t \) checks \( E \) against a known (ground) type \( t \). Algorithmic typing assumes that the term is in \( \eta \)-long form, i.e. there are no partial applications. During type checking, the outer \( \lambda \)-abstractions are handled by the checking rule C-FUN, and then the type of inner application term is inferred and compared with the given type \( b \) in C-BASE.

The only interesting case is the inference rule I-APP, which handles (uncurried) component applications using their corresponding type transformers. Nullary components are handled by the same rule (note that in this case \( \left[ c \right] = \text{fresh}(\Lambda(c)) \)). 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 a Hindley-Milner system simplified to a first-order language where the top-level type is given, so we never have to guess types of $\lambda$-binders. 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 $A_i \in B_\bot$ that contains $\tau$ and $\bot$, and is a sub-lattice of the type subsumption lattice (importantly, it is closed under $\cap$). For example, in a library with a nullary constructor $A$ and two unary constructors $L$ and $M$, $\mathcal{A}_0 = \{\tau, \bot\}$, $\mathcal{A}_1 = \{\tau, A, L, \tau, \bot\}$, and $\mathcal{A}_2 = \{\tau, A, L, L (M \tau), M (M \tau), \bot\}$ are abstract covers. Note that in a cover, the scope of a type variable is each individual base type, so the different instances of $\tau$ above are unrelated. We say that an abstract cover $\mathcal{A}'$ refines a cover $\mathcal{A}$ ($\mathcal{A}' \leq \mathcal{A}$) if $\mathcal{A}$ is a sub-lattice of $\mathcal{A}'$. In the example above, $\mathcal{A}_2 \leq \mathcal{A}_1 \leq \mathcal{A}_0$.

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

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

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 $B_\bot$, 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 $B \subseteq id(\alpha_{\mathcal{A}}(B))$ 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 $\Lambda; \Gamma \vdash_{\mathcal{A}} e \Rightarrow B$, 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 $\lambda x.s$ where $\Lambda(\lambda) = \forall \beta. L \beta \rightarrow M \beta$ and $\Gamma(xs) = L (M A)$. Then in $\mathcal{A}_1$ we infer $\Lambda; \Gamma \vdash_{\mathcal{A}_1} \lambda x.s \Rightarrow \tau$, since $\alpha_{\mathcal{A}_1}(L (M A)) = L \tau$ and $\lambda x.s \Rightarrow M \tau$, but $M \tau$ is abstracted to $\tau$. However, in $\mathcal{A}_2$ we infer $\Lambda; \Gamma \vdash_{\mathcal{A}_2} \lambda x.s \Rightarrow M (M \tau)$, since $\alpha_{\mathcal{A}_2}(L (M A)) = L (M \tau)$, and $\lambda x.s \Rightarrow M (M \tau)$, 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 $\mathcal{A}' \leq \mathcal{A}$:

**Theorem 3.4 (Typing Preservation).** If $\mathcal{A}' \leq \mathcal{A}$ and $\Lambda; \Gamma \vdash_{\mathcal{A}} E \Leftarrow t$ then $\Lambda; \Gamma \vdash_{\mathcal{A}'} E \Leftarrow t$.

As $B_\bot \leq \mathcal{A}$ for any $\mathcal{A}$, the above Theorem 3.4 implies that abstract typing conservatively over-approximates concrete typing:

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

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 \((\Lambda, t)\) 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: \(\lambda_H\) has no higher-order functions or recursion, hence any well-typed program has an equivalent \(n\)-long \(\beta\)-normal form. We treat the query type as a monotype without loss of generality: any query polytype \(\overline{\nu}.T\) is equivalent to \([\tau \mapsto C]T\) where \(C\) are fresh nullary type constructors. The synthesis problem in \(\lambda_H\) 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 Appendix A).

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 \(\Lambda; \cdot \vdash _{\mathcal{A}} E \iff t\). 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 \((\Lambda, t)\), 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 the a set of transitions, \(E:\ (P \times T) \cup (T \times P) \rightarrow \mathbb{N}\) 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 \(M:\ P \rightarrow \mathbb{N}\) 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 \(p:\ M_1(p) \geq E(p, t) \land M_2(p) = M_1(p) - E(p, t) + E(t, p)\). A sequence of transitions \(t_1, \ldots, t_n\) is a path between \(M\) and \(M’\) if \(M \xrightarrow{t_1} M_1 \ldots \xrightarrow{t_n} M’\) is a sequence of transition firings.

ATN Construction. Consider an abstract synthesis problem \((\Lambda, t, \mathcal{A})\), where \(t = b_1 \rightarrow \ldots \rightarrow b_n \rightarrow b\). An abstract transition net \(N(\Lambda, t, \mathcal{A})\) is a 5-tuple \((P, T, E, I, F)\), where \((P, T, E)\) is a Petri net, \(I:\ P \rightarrow \mathbb{N}\) 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 \(\alpha_{\mathcal{A}}([c](A_1, \ldots, A_m)) \equiv A\); set \(E(t, A) = 1\) and add 1 to \(E(A_j, t)\) for every \(j \in (1, m)\);
5. for each initial place \(\{p \in P \mid I(p) > 0\}\), add a self-loop copy transition \(\kappa\) to \(T\), setting \(E(p, \kappa) = 1\) and \(E(\kappa, p) = 2\), and a self-loop delete transition \(\delta\) to \(T\), setting \(E(p, \delta) = 1\) and \(E(\delta, p) = 0\).

Given an ATN \(N = (P, T, E, I, F)\), \(M_F\) 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 \((\pi \models N)\), 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 \(\pi\) corresponds to a set of normal-form terms \(\text{terms}(\pi)\). 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.

**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 \(N(\Lambda, t, \mathcal{A})\). Next, the function \texttt{ShortestValidPath} uses a constraint solver to find a shortest valid path \(\pi \models N^3\). 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 program has no solution, so the algorithm returns \(\perp\). Otherwise, it enumerates all programs \(E \in \text{terms}(\pi)\) and type-checks them abstractly, until it encounters an \(E\) that is abstractly well-typed (such an \(E\) must exists per Theorem 4.3).

Our ATN construction is inspired by but different from the approach used by SyPet [Wang et al. 2018]. 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 \(\alpha_{\mathcal{A}}([c] (A_1, \ldots, A_m))\). This result is either \(\perp\), in which case no transition is added, or some \(A \in P\), in which case we add a transition from \(A_1, \ldots, A_m\) to \(A\).

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

**Theorem 4.2 (ATN Completeness).** If \(\Lambda; \cdot \vdash_{\mathcal{A}} E \leftarrow t\) and \(E \in \text{terms}(\pi)\) then \(\pi \models N(\Lambda, t, \mathcal{A})\).

**Theorem 4.3 (ATN Soundness).** If \(\pi \models N(\Lambda, t, \mathcal{A})\), then \(\exists E \in \text{terms}(\pi)\) s.t. \(\Lambda; \cdot \vdash_{\mathcal{A}} E \leftarrow t\).

**Enforcing Relevance.** Finally, consider copy transitions \(\kappa\) and delete transitions \(\delta\): in this section, we describe an ATN that implements a simple, structural type system, where each function argument can be used zero or more times. Hence we allow the ATN to duplicate tokens in the initial marking \(I\) using \(\kappa\) transitions and discard them using \(\delta\) transitions. We can easily adapt the ATN definition to implement a relevant type system by eliminating the \(\delta\) transitions (this is what our implementation does, see Sec. 5.3); a linear type system can be supported by eliminating both.

---

3Sec. 5.3 details our encoding of ATN reachability into constraints.
4.2 The TYGAR Algorithm

The abstract synthesis algorithm from Fig. 7 either returns ⊥, 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 A (starting with some 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 A_0, and either returns a solution E to the synthesis problem or ⊥ 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 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 ⊥. 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 A and a spurious program E and returns a refinement A' of the current cover (A' ≤ A) such that E is abstractly ill-typed in A' (Λ; · ⊢ₐ E ↔ 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 A_0 has no influence on the correctness of the algorithm. A natural choice is the most general cover A_T = {τ, ⊥}. In our experiments (Sec. 6) we found that synthesis is more efficient if we pick the initial cover A_Q(b_i → b) = close((τ, b_i, b, ⊥))⁴, which represents the query type t = b_i → b concretely. Intuitively, the reason is that the distinctions between the types in t are very likely to be important for solving the synthesis problem, so there is no need to make the algorithm re-discover them from scratch.

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

Theorem 4.4 (Soundness). If SYNTHESIZE(Λ, t, A_0) returns E then Λ; · ⊢_{ₐ} E :: t.

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

Theorem 4.5 (Completeness). If ∃E. Λ; · ⊢_{ₐ} E :: t then SYNTHESIZE(Λ, t, A_0) returns some E′ ≠ ⊥.

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 ⊥ or a program E that is larger than E_0, since E_0 is abstractly well-typed at any 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.

⁴Here close(A) closes the cover under meet, as required by the definition of sublattice.
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 \textsc{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 \((\Lambda; \vdash E \leftarrow t)\), but well-typed abstractly \((\Lambda; \vdash \mathcal{A} E \leftarrow t)\). It produces a refinement of the cover \( \mathcal{A}' \leq \mathcal{A} \), such that \( E \) is ill-typed abstractly in that new cover \((\Lambda; \vdash \mathcal{A}' E \leftarrow t)\).

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

Let us now explain how \( P \) is constructed. Let \( E = \lambda x_i : b_1 \to t \), and \( \Gamma = \bar{x}_i : \bar{b}_1 \). There are two reasons why \( E \) might not type-check against \( t \): either \( e_{\text{body}} \) on its own is ill-typed or it has a non-bottom type that nevertheless does not subsume \( b \). To unify these two cases, \textsc{Refine} constructs a new application term \( e^\ast = r e_{\text{body}} \), where \( r \) is a dedicated component of type \( b \to b \); such \( e^\ast \) is guaranteed to be ill-typed on its own: \( \Lambda; \Gamma \vdash e^\ast \Rightarrow \bot \). Lines 4–5 initialize \( P \) for each subterm of \( e^\ast \) with the result of concrete type inference. At this point \( P \) already constitutes a valid proof of untypeability, but it contains too much information; in line 6 we the call to \textsc{Generalize} removes as much information from \( P \) as possible while maintaining enough to prove that \( e^\ast \) is ill-typed. More precisely, \textsc{Generalize} maintains three crucial invariants that together guarantee that \( P \) is a proof of untypeability:

\begin{itemize}
  \item \( I_1 \): \( (P \text{ subsumes concrete typing}) \) For any \( e \in \) subterms\( (e^\ast) \), if \( \Lambda; \Gamma \vdash e \Rightarrow B \), then \( B \subseteq P[e] \);
  \item \( I_2 \): \( (P \text{ abstracts type transformers}) \) For any application subterm \( e = c e_j \), \( [c](\bar{P}[e_j]) \subseteq P[e] \);
  \item \( I_3 \): \( (P \text{ proves untypeability}) P[e^\ast] = \bot \).
\end{itemize}

**Lemma 4.6.** If \( I_1 \land I_2 \land I_3 \) then \( P \) is a proof of untypeability: if \( \text{range}(P) \subseteq \mathcal{A}' \) then \( \Lambda; \vdash \mathcal{A}' E \leftarrow t \).

**Proof Sketch.** We can show by induction on the derivation that for any \( \mathcal{A}' \supseteq \text{range}(P) \) and node \( e \), \( \Lambda; \Gamma \vdash \mathcal{A}' e \Rightarrow B \subseteq P[e] \) (base case follows from \( I_1 \), and inductive case follows from \( I_2 \)). Hence, \( \Lambda; \Gamma \vdash \mathcal{A}' e^\ast \Rightarrow B \subseteq P[e^\ast] = \bot \) (by \( I_3 \)), so \( \Lambda; \Gamma \vdash \mathcal{A}' e_{\text{body}} \Rightarrow B \not\subseteq b \), and \( \Lambda; \vdash \mathcal{A}' E \leftarrow t \).

**Correctness of \textsc{Generalize}**. Now that we know that invariants \( I_1 \sim I_3 \) are sufficient for correctness, let us turn to the inner workings of \textsc{Generalize}. This function starts with the initial proof \( P \) (concrete typing), and recursively traverses the term \( e^\ast \) top-down. At each application node \( e = c e_j \) it weakens the argument labels \( \bar{P}[e_j] \) (lines 4–7). The weakening step performs lattice search to find more general values for \( \bar{P}[e_j] \) allowed by \( I_2 \). More concretely, each new value \( B_j \) starts out as
the initial value of $P[e_j]$; 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 $[c](\overline{B_j}) \subseteq P[e]$. 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) $P[e_j] \subseteq \overline{B_j}$ and (2) $[c](\overline{B_j}) \subseteq P[e]$.

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

**Example 1.** Let us walk through the refinement step in iteration 2 of our running example from Sec. 2.2. As a reminder, $\Lambda(t) = \forall \alpha. \alpha \rightarrow M \alpha \rightarrow \alpha$ and $\Lambda(\bot) = \forall \beta. \bot \rightarrow M \beta$. Consider a call to `Refine($A,E,t$)`, where $A = \{r, A, L, \tau, \bot\}$, $E = \lambda x_1 x_2. f \ x_1 \ (l \ x_2)$ and $t = A \rightarrow L (M A) \rightarrow A$. Let us denote $\Gamma = x_1 : A, x_2 : L (M A)$. It is easy to see that $E$ is ill-typed concretely but well-typed abstractly, since, as explained above, $\Lambda; \Gamma \uparrow_{A} l x_2 \Rightarrow \tau$, and hence $\Lambda; \Gamma \uparrow_{A} f x_1 (l \ x_2) \Rightarrow A$. `Refine` first constructs $e^* = r \ e_{\text{body}}$; the AST for this term is shown on Fig. 9 (left). It then initializes the mapping $P$ with concrete inferred types, depicted as red labels; as expected $P[e^*] = \bot$. The blue labels show $P'$ obtained by calling `Generalize` through the following series of recursive calls:

- In the initial call to `Generalize`, the term $e$ is $r \ e_{\text{body}}$; although it is an application, we do not weaken the label for $e_{\text{body}}$ since its concrete type is $\bot$, which cannot be weakened.
- We move on to $e_{\text{body}} = f \ x_1 \ l$ with $P[x_1] = A$ and $P[l] = M (M A)$. The former type cannot be weakened: an attempt to replace $A$ with $\tau$ causes $\overline{f}$ to produce $M A \not\subseteq \bot$. The latter type can be weakened by replacing $A$ with $\tau$ (since $[\overline{f}]A (M (M \tau)) = \bot$), but no further.
- The first child of $f$, $x_1$, is a variable so $P$ remains unchanged.
- For the second child of $f$, $l = l \ x_2$, $l$’s signature allows us to weaken $P[x_2]$ to $L (M \tau)$ but no further, since $[\overline{l}]L (M \tau) = M (M \tau)$ but $[\overline{l}]L \tau \subseteq M (M \tau)$.
- 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 $A$ with three type constructors, $Z$, $U$, and $B$ (with arities 0, 1, and 2, respectively), and two components, $f$ and $g$, such that: $\Lambda(f) = \forall \alpha. B \alpha \alpha$ and $\Lambda(g) = \forall \beta. B \ (U \beta) \beta \rightarrow Z$. Consider the synthesis problem $(\Lambda, Z)$, which has no solutions. Assume that the initial abstract cover is $\mathcal{A}_0 = \{\tau, \bot\}$, as shown in the upper left of Fig. 10. `SynAbstract($\Lambda, Z, \mathcal{A}_0$)`
returns a program \( f \), which is spurious, hence we invoke \( \text{REFINE}(A_0, f, Z) \). The results of concrete type inference are shown as red labels in Fig. 10; in particular, note that because \( f \) is a nullary component, \( \llbracket f \rrbracket \) is simply a fresh instance if its type, here \( B \tau \tau \), which can be generalized to \( B \alpha \beta \): the root cause of the type error is that \( \tau \) does not accept a \( B \). In the second iteration, \( A_0 = \{ \tau, B \alpha \beta, \bot \} \) and \( \text{SYNABSTRACT}(\Lambda, Z, A_1) \) returns \( g \ f \), which is also spurious. In this call to \( \text{REFINE} \), 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 different arguments. Adding \( B \tau \tau \) to the cover, results in the ATN on the right, which does not have a valid path (i.e. \( \text{SYNABSTRACT} \) 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: \( \forall \alpha \cdot \Gamma \vdash f \equiv B \tau \tau \). (2) \( \text{SYNABSTRACT} \) can sometimes detect that there is no solution, even when the space of all possible ground base types is infinite. (3) To prove untypeability of \( g \ f \), our abstract domain must be able to express non-linear type-level terms (i.e. types with repeated variables, like \( B \tau \tau \)); we could not, for example, replace type variables with a single construct \( ? \), as in gradual typing [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 \( \lambda_H \), 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 \( \lambda_H \). Next, we give more detail on how H+ translates a Haskell synthesis problem \((\Lambda, I)\) into a \( \lambda_H \) synthesis problem \((\Lambda, I)\):

(1) \( \Lambda \) includes a fresh binary type constructor \( \mathcal{F} \alpha \beta \) (used to represent function types).

(2) Every declaration of type class \( C \tau \) with methods \( m_i :: \forall \tau. T_i \) in \( \hat{\Lambda} \) gives rise to a type constructor \( \mathcal{C} \tau \) (the dictionary type) and components \( m_i :: \forall \tau. \mathcal{C} \tau \rightarrow T_i \) in \( \Lambda \). For example, a type class declaration \( \text{class} \ Eq \alpha \) where (\( \equiv \)) : : a \rightarrow a \rightarrow Bool \ creates a fresh type constructor \( \mathcal{E}q \alpha \) and a component (\( \equiv \)) : : \mathcal{E}q\theta \alpha \rightarrow \alpha \rightarrow \alpha \rightarrow Bool \).

(3) Every instance declaration \( C B \) in \( \hat{\Lambda} \) produces a component that returns a dictionary \( \mathcal{C} \). So \( \text{instance} \ Eq \ Int \) creates a component eqInt : : \mathcal{E}q\theta \ Int \), while a subclass instance like \( \text{instance} \ Eq \ a \Rightarrow \mathcal{E}q \) [a] creates a component eqList : : \mathcal{E}q\theta \ a \rightarrow \mathcal{E}q\theta \ [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 \( \hat{\Lambda} \), we add a component \( \hat{c} \) to \( \Lambda \) and define \( \hat{\Lambda}(c) = \text{desugar} \left( \hat{\Lambda}(c) \right) \), where the translation function desugar, which eliminates type class constraints and higher-order types, is defined as follows:

\[
\text{desugar} (\forall \tau. (C_1 \tau_1, \ldots, C_n \tau_n) \Rightarrow T) = \forall \tau. \mathcal{C} \tau_1 \rightarrow \cdots \rightarrow \mathcal{C} \tau_n \rightarrow \text{desugar}(T)
\]

\[
\text{desugar}(T_1 \rightarrow T_2) = \text{base}(T_1) \rightarrow \text{desugar}(T_2) \quad \text{desugar}(B) = B
\]

\[
\text{base}(T_1 \rightarrow T_2) = \mathcal{F} \text{base}(T_1) \text{base}(T_2) \quad \text{base}(B) = B
\]
For example, Haskell components on the left are translated into $\lambda_H$ components on the right:

- `member :: Eq 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 $\Lambda$, we add a nullary component $c'$ to $\Lambda$ and define $\Lambda(c') = \text{base}(\Lambda(c))$. For example: `any' :: F (F α Bool) (F [α] Bool)`.

(6) Finally, the $\lambda_H$ query type $t$ is defined as desugar($t$).

**Limitations.** In modern Haskell, type classes often constrain *higher-kindred* 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-kindred type variables is beyond the scope of this paper. In theory our encoding with nullary components (Sec. 2.4) is complete for higher-order functions 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 ($\$ :: F α β → α → β$) 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 $\mathcal{A}' = \mathcal{A} \cup \{A_{\text{new}}\}$ 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 $A_{\text{new}}$ in the $\subseteq$ partial order; for example, in the cover $\{\tau, \mathcal{P} \alpha \beta, \mathcal{P} \alpha \mathcal{B}, \mathcal{P} \alpha \mathcal{B}, \mathcal{P} \mathcal{A} \mathcal{B}, \bot\}$, the parents of $\mathcal{P} \alpha \mathcal{B}$ are $\{\mathcal{P} \alpha \beta, \mathcal{P} \alpha \mathcal{B}\}$. Intuitively, adding $A_{\text{new}}$ 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 $\alpha_{\mathcal{A}} \left(\left[c\right](\overline{A_i})\right) = A$ such that $A \in \text{parents}$; if $\alpha_{\mathcal{A}} \left(\left[c\right](\overline{A_i})\right) = A_{\text{new}}$, set $E(t, A) = 0$ and $E(t, A_{\text{new}}) = 1$.
2. Consider a transition $t \in T$ that represents the abstract instance $\alpha_{\mathcal{A}} \left(\left[c\right](\overline{A_i})\right) = A$ such that at least one $A_i \in \text{parents}$; consider $\overline{A'}_i$ obtained from $\overline{A_i}$ by substituting at least one $A_i \in \text{parents}$ with $A_{\text{new}}$; if $\alpha_{\mathcal{A}} \left(\left[c\right](\overline{A'})\right) = A' \neq \bot$, 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 $l$ has 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 \(\phi\) that encodes all valid paths of a given length \(\ell\); the overall search will then proceed by iteratively increasing the length \(\ell\). We encode the state of each place \(p \in P\) at each time step \(t \in [0, \ell]\) as an integer variable \(p_t\). We assign each transition in \(T\) a unique integer id \(t\). We encode the transition firing at each time step \(t \in [0, \ell]\) as an integer variable \(f_t\) so that \(f_t = t\) indicates that the transition \(t\) is fired at time step \(t\). 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 \(\phi\) is a conjunction of the following constraints:

1. At each time step, a valid transition is fired: \(\bigwedge_{t=0}^{\ell-1} 1 \leq f_t \leq |T|\)
2. If a transition \(t\) is fired at time step \(t\) then all places \(p \in \text{pre}(t)\) have sufficiently many tokens: \(\bigwedge_{t=0}^{\ell-1} \bigwedge_{t=1}^{|T|} f_t = t \implies \bigwedge_{p \in \text{pre}(t)} p_t \geq E(p, t)\)
3. If a transition \(t\) is fired at time step \(t\) then all places \(p \in \text{pre}(t) \cup \text{post}(t)\) will have their markings updated at time step \(t+1\): \(\bigwedge_{t=0}^{\ell-1} \bigwedge_{t=1}^{|T|} f_t = t \implies \bigwedge_{p \in \text{pre}(t) \cup \text{post}(t)} p_{t+1} = p_t - E(p, t) + E(t, p)\)
4. If none of the outgoing or incoming transitions of a place \(p\) are fired at time step \(t\), then the marking in \(p\) does not change: \(\bigwedge_{t=0}^{\ell-1} \bigwedge_{p \in P} (\bigwedge_{t \in \text{pre}(p) \cup \text{post}(p)} \neg f_t = t) \implies p_t = p_{t+1}\)
5. The initial marking is \(I\): \(\bigwedge_{p \in P} p_0 = I(p)\).
6. The final marking is valid: \(\bigvee_{f \in F} (f_\ell = 1 \land \bigwedge_{p \in P \setminus \{f\}} p_\ell = 0)\).

**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 \(f_\ell = 1\) (and \(p_\ell = 0\) 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: \(\bigwedge_{p \in P} (|I(p)| > 0) \land p_0 = I(p)\) and \(\bigwedge_{p \in P} (|I(p)| = 0) \land p_0 = 0\). 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 a common property among desirable programs.

6 EVALUATION

Next, we describe an empirical evaluation of two research questions of H+:
• Efficiency: Is TYGAR an efficient synthesis strategy?
• Quality of Solutions: Are the synthesized code snippets interesting?

Component library. We use a single component in all experiments. To create this set, we picked 291 popular functions from common modules of Haskell’s base and bytestring packages. These modules include 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. These modules feature type classes and polymorphism, to support many different applications. Out of 291 components, we found 7 that adversely affected the quality for solutions, because they are highly polymorphic but are never applied productively, so we disabled their application (they can still be used as arguments to higher-order functions). These include id, const, fix, on, flip, &,. (.).

Query Selection. The complete set of benchmark queries is found in the second column of Fig. 11. We collected queries from three sources (1) Queries made to Hoogle between January 2015 and February 2019. Of the 3,779,571 queries and 2,243 queries occurring at least five times, many were syntactically invalid (e.g. “a →”), uninhabited (e.g. “a → b”), or incompatible with our module set. This left us with 24 Hoogle benchmarks. (2) We searched Stackoverflow in May and June 2019 for common Haskell queries, producing 6 different queries (3) The authors compiled 17 queries from their own experience. These three produce a total of 44 benchmarks.

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.

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 1 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 \( \mathcal{A}_T = \{ \tau, \bot \} \).

(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 slow, 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
Fig. 11. H+ synthesis times and solution quality on 44 benchmarks. Suffixes ‘-QB10’, ‘-Q’, ‘-0’, ‘-NO’ correspond to four variants of the search algorithm: TYGAR-QB [10], TYGAR-Q, TYGAR-0, and NOGAR. Prefixes ‘t-’, ‘st-’, and ‘tc-’ denote, respectively, the total time to a first solution, time spend in the SMT solver, and time spent type checking (including demand analysis). All times are in seconds. Absence indicates no interesting solution was in the first five with H+. ‘H-D’ is the number of interesting solutions in the first five on H+. ‘H-R’ is the same number with the demand analyzer disabled. ‘H-R’ is the same number with structural typing over relevant typing. Absence indicates no interesting solution was in the first five.

NOGAR could not, for example, for `appBoth` :: (a → b) → (a → c) → a → (b, c). We conclude that refinement seems to be able to discover interesting abstractions, but after a while it starts 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 corresponds 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 one disposal renders searches faster: the bounds of 10 and 15 seem to offer a sweet spot. In particular TYGAR-QB [10] solves the most queries—43 out of 44—and the median synthesis time is 1.4 seconds. In the rest of this section we use TYGAR-QB [10] as the default H+ configuration.
6.2 Quality of Solutions

Setup. To evaluate the quality of the solutions, we ask H+ to return the first five well-typed results for each query. Complete results are available in Appendix B. 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\(^5\). Fig. 11 reports for each query, the number of the interesting solution among the five. In this experiment, we use a timeout of 100 seconds to produce all five solutions. 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 ATN is not required to use all parameters to the input query).

Analysis. The average number of interesting solutions per query for H+ was 1.6. H-D had an average number of an interesting solution of 1.5. Lastly, H-R produced on average 0.6 interesting results. Whenever a solution was found without the demand analyzer or without relevant typing, the unhindered version would find the same number or more interesting solutions. We observe that on easy queries—taking less then 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. Anecdotally, we found that the density of interesting solutions was higher with the demand analyzer, that is, while we may have gotten fewer solutions, they were of a higher caliber. Further, in our experience, the demand analyzer was most useful when queries involved types like Either a b, where one could produce a value of type a with a value of type b by constructing and destructing the Either.

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.

\(^5\)Unfortunately, we do not have ground truth solutions for most of our queries, so we have to resort to subjective analysis.
\begin{itemize}
  \item \((a \rightarrow a) \rightarrow a \rightarrow \text{Int} \rightarrow a\) corresponds to benchmark 11 (applyN Times); the solution from Sec. 2 is generated at rank 10.
  \item Eq \(a \Rightarrow [(a, b)] \rightarrow a \rightarrow b\) corresponds to benchmark 18 (lookup); the solution from Sec. 2 is generated at rank 1.
\end{itemize}

H+ has also produced code snippets that surprised us: for example, on the query \((a \rightarrow b, a) \rightarrow b\), the authors’ intuition was to destruct the pair then apply the function. Instead H+ produces \(\lambda x \rightarrow \text{uncurry } ($) \ x\) or alternatively \(\lambda x \rightarrow \text{uncurry id } x\), both of which, contrary to our intuition, are not only well-typed, but also are functionally equivalent to our intended solution. It was welcome to see a synthesis tool write more succinct code that its authors.

\section{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].

\textbf{API Search.} Modern IDEs support various forms of code-completion, based on at the very least common prefixes of names (e.g. completing \texttt{In} into \texttt{Integer} or \texttt{fo} into \texttt{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 \textsc{Hoogle} [Mitchell 2004] that search for type isomorphisms [Di Cosmo 1993] to find a functions that “match” a given type signature (query). The above can be viewed as returning single-token results (e.g. a single function or field), as opposed to our goal of searching for terms that combine components in order to satisfy a given type query.

\textbf{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 occuring code snippets and hence, for understanding how to use some new API, they are less so for synthesizing terms that implement highly structured type-specifications.

\textbf{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 \textsc{Djinn} [Augusston 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 \textit{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 an output type on lists, \textsc{Djinn} returns functions that return 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. \textsc{Synquid} [Polikarpova et al. 2016] and \textsc{Myth2} [Frankle et al. 2016] use different flavors of refinement types to synthesize recursive functions, while \textsc{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 due to the combinatorial explosion in the search space.
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).

8 CONCLUSIONS
We have presented TYGAR, a new algorithm for synthesizing terms over polymorphic components that inhabit a given query type. The key challenge here is the infinite space of monomorphic instances arising from the polymorphic signatures. We introduced a new notion of abstract typing

that allows us to use ideas from the framework of abstract interpretation to compute a finite overapproximation of this search space. We then show how spurious terms that are well-typed in the abstract domain but ill-typed in reality, yield proofs of untypeability which can then iteratively refine the abstract search space until a well-typed solution is found.

We have implemented TYGAR in H+, and our evaluation on a suite of 44 queries demonstrates the benefits of counterexample-driven refinement. In particular, we show how lazily refining a coarse abstract domain using proofs of untypeability allows us to synthesize correct results faster than a naive approach that eagerly instantiates all the types from the query followed by a brute-force enumeration. Our experiments further demonstrate that the gains from iterative refinement over enumeration are even more pronounced on harder queries over more complex types. Our support for polymorphism also allows H+ to work with higher-order and type-class constrained components, which, thanks to parametricity, allows for more precise queries than simple monomorphic types.

In future work it would be valuable to investigate ways to improve the quality of the results, e.g. by prioritizing components that are more popular, or less partial, or by extending our method to use other forms of specifications such as examples or refinement types.

REFERENCES


Program Synthesis by Type-Guided Abstraction Refinement

1:27


Tihomir Oveto, Kvero Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete completion using types and weights. In PLDI.


Fig. 14. Uncurried version of declarative typing.

A PROOFS

A.1 Type Transformers

LEMMA A.1 (Monotonicity of Substitution). If $\sigma' \subseteq \sigma$ then $\sigma'B \subseteq \sigma B$.

LEMMA A.2 (Monotonicity of Unification). If $B_1 \subseteq B_2$ then $\text{mgu}(B, B_1) \subseteq \text{mgu}(B, B_2)$

LEMMA A.3 (Monotonicity of Type Transformers). For any component $c$ and any types $B_i^1$ and $B_i^2$, such that $B_i^1 \subseteq B_i^2$, we have $[c](B_i^1) \subseteq [c](B_i^2)$.

PROOF. Let $\text{fresh}(\Lambda(c)) = B_i \rightarrow B$. By the definition of type transformers, we have

\[
[c](B_i^1) = \sigma_1 B_i \quad \sigma_1 = \text{mgu}(B_i, B_i^1)
\]

\[
[c](B_i^2) = \sigma_2 B_i \quad \sigma_2 = \text{mgu}(B_i, B_i^2)
\]

Since $B_i^1 \subseteq B_i^2$, we have $\sigma_1 \subseteq \sigma_2$ by Theorem A.2, and hence $\sigma_1 B_i \subseteq \sigma_2 B_i$ by Theorem A.1. \qed

LEMMA A.4 (Soundness of Type Transformers). If $\text{fresh}(\Lambda(c)) = B_i \rightarrow B$ and $\sigma B_i \subseteq B_i^2$ then $\sigma B \subseteq [c](B_i^2)$.

PROOF. By Theorem A.3, since $\sigma B_i \subseteq B_i^2$, we have $[c](\sigma B_i) \subseteq [c](B_i^2)$. But $[c](\sigma B_i) = \sigma B$, because $\text{mgu}(B, \sigma B) \equiv \sigma$. Hence $\sigma B \subseteq [c](B_i^2)$ as desired. \qed

A.2 Algorithmic Typing

In the following, we use a version the declarative type system in Fig. 14, which is defined over uncurried applications, and hence more closely matches algorithmic typing. It is straightforward to show that for terms in $\eta$-long form, this type system is equivalent to the one in Fig. 5.

LEMMA A.5 (Soundness of inference). If $\Lambda; \Gamma \vdash e \Rightarrow B$ and $b \sqsubseteq B$, then $\Lambda; \Gamma \vdash e :: b$.

PROOF. By induction on the derivation of $\Lambda; \Gamma \vdash e \Rightarrow B$.

- Case I-VAR: Given the conclusion $\Lambda; \Gamma \vdash x \Rightarrow b'$, we get to assume $\Gamma(x) = b'$. Consequently, applying T-VAR, we get $\Lambda; \Gamma \vdash x :: b'$. But since $b'$ is ground and $b \sqsubseteq b'$, we have $b = b'$.

- Case I-APP: Given the conclusion $\Lambda; \Gamma \vdash c \overline{c_i} \Rightarrow [c](B_i)$, we get to assume $\Lambda; \Gamma \vdash e_i \Rightarrow B_i$. Assume $\text{fresh}(\Lambda(c)) = B_i' \rightarrow B'$. By definition of $[c]$ we have $[c](B_i) = \sigma_1 B'$ where $\sigma_1 = \text{mgu}(B_i', B_i)$. By the assumption of the theorem $b \subseteq B = [c](B_i)$, and hence there exists $\sigma_2$ such that $b = \sigma_2([c](B_i)) = (\sigma_2 \circ \sigma_1)B'$. Let $\rho$ be some substitution such that $(\rho \circ \sigma_2 \circ \sigma_1)B'_i$ is ground for all for all $B'_i$; we will denote these ground types $b_i$. Then each $b_i = (\rho \circ \sigma_2)(\sigma_1 B'_i)$ (since $\sigma_1$ is a unifier), and hence $b_i \subseteq B_i$. Then by the IH, we can show for all $e_i: \Lambda; \Gamma \vdash e_i :: b_i$ (1).
Finally, let us use T-App to construct the derivation of $\Lambda; \Gamma \vdash c \; \overline{e}_i :: b$. The middle premise follows with $\sigma = \rho \circ \sigma_2 \circ \sigma_1$ (note that $\sigma B' = \rho b = b$, since $b$ is already ground). We get the last premise by (1).

\[ \square \]

**Lemma A.6 (Completeness of Inference).** If $\Lambda; \Gamma \vdash e :: b$, then $\exists B. \Lambda; \Gamma \vdash e \Rightarrow B \land b \sqsubseteq B$.

**Proof.** By induction on the derivation of $\Lambda; \Gamma \vdash e :: b$.

- Case T-Var: Given the conclusion $\Lambda; \Gamma \vdash x :: b$, we get to assume $\Gamma(x) = b$. Using I-Var we build the derivation $\Lambda; \Gamma \vdash x \Rightarrow b$. Trivially $b \sqsubseteq b$ by reflexivity of $\sqsubseteq$.
- Case T-App: Given the conclusion $\Lambda; \Gamma \vdash c \; \overline{e}_i :: b$, we get to assume for each $i$: $\Lambda; \Gamma \vdash e_i :: b_i$, where $\overline{b}_i \rightarrow b = \sigma T$ and $T = \overline{B}_i \rightarrow B' = \text{fresh}(\Lambda(c))$. By IH we obtain $\overline{B}_i$, such that $\Lambda; \Gamma \vdash e_i \Rightarrow B_i$ and $b_i \sqsubseteq B_i$, for each $i$. Using I-Var we build the derivation $\Lambda; \Gamma \vdash c \; \overline{e}_i \Rightarrow \{c\}(\overline{B}_i)$. It remains to show that $b \sqsubseteq \{c\}(\overline{B}_i)$.
  
  Since $\overline{b}_i \subseteq \overline{B}_i$ and $b_i = \sigma B_i'$, we get $\sigma B_i' \sqsubseteq \overline{B}_i$. Hence we can apply Theorem A.4 to conclude $\sigma B' \sqsubseteq \{c\}(\overline{B}_i)$, but $\sigma B' = b$, so we obtain the desired conclusion.
- Case T-Fun: impossible since $b$ is a base type.

\[ \square \]

**Lemma A.7 (Soundness of Checking).** If $\Lambda; \Gamma \vdash E \trianglelefteq t$, then $\Lambda; \Gamma \vdash E :: t$.

**Proof.** By induction on the derivation of $\Lambda; \Gamma \vdash E \trianglelefteq t$.

- Case C-Base: Given the conclusion $\Lambda; \Gamma \vdash e \trianglelefteq b$, we get to assume $\Lambda; \Gamma \vdash e \Rightarrow B$ and $b \sqsubseteq B$. By Theorem A.5, we directly obtain $\Lambda; \Gamma \vdash e :: b$.
- Case C-Fun: Given the conclusion $\Lambda; \Gamma \vdash \lambda x. E \trianglelefteq b \rightarrow t$, we get to assume $\Lambda; \Gamma, x : b \vdash E \trianglelefteq t$. By IH, we get $\Lambda; \Gamma, x : b \vdash E :: t$. Hence we use T-Fun to obtain $\Lambda; \Gamma \vdash \lambda x. e :: b \rightarrow t$.

\[ \square \]

**Lemma A.8 (Completeness of Checking).** If $\Lambda; \Gamma \vdash E :: t$ then $\Lambda; \Gamma \vdash E \trianglelefteq t$.

**Proof.** By induction on the derivation of $\Lambda; \Gamma \vdash E :: t$.

- Case T-Var, T-App: Given $\Lambda; \Gamma \vdash e :: b$, we use Theorem A.6 to derive $\Lambda; \Gamma \vdash e \Rightarrow B$ and $b \sqsubseteq B$. Hence we apply C-Base to obtain $\Lambda; \Gamma \vdash e :: b$.
- Case T-Fun: Given the conclusion $\Lambda; \Gamma \vdash \lambda x. E :: b \rightarrow t$ we get to assume $\Lambda; \Gamma, x : b \vdash E :: t$. By IH $\Lambda; \Gamma, x : b \vdash E \trianglelefteq t$. We apply C-Fun to obtain $\Lambda; \Gamma \vdash \lambda x. e :: b \rightarrow t$.

\[ \square \]

**Theorem A.9 (Type Checking is Sound and Complete).** $\Lambda; \cdot \vdash E :: t$ iff $\Lambda; \cdot \vdash E \trianglelefteq t$.

**Proof.** By Theorem A.7 and Theorem A.8.

\[ \square \]

**A.3 Abstract Typing**

**Lemma A.10 (Refinement).** If $B' \sqsubseteq B$ and $\mathcal{A}' \subseteq \mathcal{A}$, then $\alpha_{\mathcal{A}'}(B') \subseteq \alpha_{\mathcal{A}}(B)$.

**Proof.** Let $A = \alpha_{\mathcal{A}}(B)$, $A' = \alpha_{\mathcal{A}'}(B')$. Because $\mathcal{A} \subseteq \mathcal{A}'$, we have $A \in \mathcal{A}'$. Since $B' \subseteq B \sqsubseteq A$, we have $A' \subseteq A$ by definition of $\alpha_{\mathcal{A}'}$.

\[ \square \]

**Lemma A.11 (Inference Refinement).** If $\mathcal{A}' \subseteq \mathcal{A}$ and $\Lambda; \Gamma \vdash_{\mathcal{A}'} e \Rightarrow B'$, then $\Lambda; \Gamma \vdash_{\mathcal{A}} e \Rightarrow B$ and $B' \subseteq B$.

**Proof.** By induction on the derivation of $\Lambda; \Gamma \vdash_{\mathcal{A}'} e \Rightarrow B'$.
The query type \( \forall \alpha \beta. P \alpha \beta \rightarrow P \alpha \beta \) is undecidable. 

Proof. By induction on the derivation of \( \lambda; \Gamma \vdash \alpha \beta \). 

- Case C-Base: Given the conclusion \( \lambda; \Gamma \vdash \alpha \beta \) \( \equiv b \), we get to assume \( \lambda; \Gamma \vdash \alpha \beta \) \( \equiv B' \) and \( b \equiv B' \). Then by Theorem A.11 we get \( \lambda; \Gamma \vdash \alpha \beta \) \( \equiv B \) and \( B' \equiv B \), hence \( b \equiv B \). We use C-Base to conclude \( \lambda; \Gamma \vdash \alpha \beta \) \( \equiv b \).
- Case C-Fun: Given the conclusion \( \lambda; \Gamma \vdash \alpha \beta \) \( \equiv \lambda x. E \) \( \equiv b \rightarrow t \) we get to assume \( \lambda; \Gamma \vdash x : \alpha \beta \) \( \equiv \lambda x. E \) \( \equiv t \). By IH we get \( \lambda; \Gamma \vdash x : \alpha \beta \) \( \equiv \lambda x. E \) \( \equiv b \rightarrow t \).

A.4 Synthesis

Theorem A.13. The synthesis problem in \( \lambda_\text{H} \) is undecidable.

Proof. By reduction from the Post Correspondence Problem (PCP). Let \([ (a_1, b_1), \ldots, (a_n, b_n) ] \) be an instance of the PCP, where each \( a_i, b_i \) are bit strings. We translate this instance into a synthesis problem (\( \lambda, T \)) as follows.

The set of type constructors in \( \lambda \) is: two nullary constructors \( \text{Start} \) and \( \text{Goal} \), two unary constructors \( T \) and \( F \), and a single binary constructor \( P \). Given a bit string \( bs \) and a type \( T \), the type \( \text{wrap}(T, bs) \) is defined as follows:

\[
\begin{align*}
\text{wrap}(T, []) &= T \\
\text{wrap}(T, 0 : bs) &= \text{wrap}(F T, bs) \\
\text{wrap}(T, 1 : bs) &= \text{wrap}(T T, bs)
\end{align*}
\]

Now we can define the set of components in \( \lambda \):

- for each pair \( (a_i, b_i) \), there is a component \( s_i \) of type \( \text{Start} \rightarrow P \text{wrap}(\text{Start}, a_i) \text{wrap}(\text{Start}, b_i) \);
- for each pair \( (a_i, b_i) \), there is a component \( n_i \) of type \( \forall \alpha \beta. P \alpha \beta \rightarrow P \text{wrap}(\alpha, a_i) \text{wrap}(\beta, b_i) \);
- a component \( f \) of type \( \forall \alpha. P \alpha \alpha \rightarrow \text{Goal} \).

The query type \( T \) is \( \text{Start} \rightarrow \text{Goal} \). The sequence of \( P \) instances in the solution to this synthesis problem corresponds to the solution to the PCP.

For example, a PCP instance \([ (1, 101), (10, 00), (011, 11) ] \) gives rise to the following components:

\[
\begin{align*}
s1 &:: \text{Start} \rightarrow P (T \text{Start}) (T (F (T \text{Start}))) \\
s2 &:: \text{Start} \rightarrow P (T (F \text{Start})) (F (F \text{Start})) \\
s3 &:: \text{Start} \rightarrow P (T (F \text{Start})) (T (T \text{Start})) \\
n1 &:: P \alpha \beta \rightarrow P (T \alpha) (T (F (T \beta))) \\
n2 &:: P \alpha \beta \rightarrow P (F (T \alpha)) (F (F \beta)) \\
n3 &:: P \alpha \beta \rightarrow P (T (T (F \alpha))) (T (T \beta)) \\
f &:: P \alpha \alpha \rightarrow \text{Goal}
\end{align*}
\]

The solution for this instance is \( \lambda x \rightarrow f (n3 (n2 (n3 (s1 x)))) \).
A.5 Proof of Untypeability

For a $P: e \rightarrow B_\bot$ and a term $e^*$, define

$I_1$: For any $e \in \text{subterms}(e^*)$, if $\Lambda; \Gamma \vdash e \Rightarrow B$, then $B \subseteq P[e]$;
$I_2$: For any $e = c \overline{e} \in \text{subterms}(e^*)$, $[c](P[e_i]) \subseteq P[e]$;
$I_3$: $P[e^*] = \bot$.

**Lemma A.14 (Precision of Inference).** If $I_1 \land I_2$, range$(P) \subseteq \mathcal{A}$, $e \in \text{subterms}(e^*)$, and $\Lambda; \Gamma \vdash e \Rightarrow B$, then $B \subseteq P[e]$.

**Proof.** By induction on the derivation of $\Lambda; \Gamma \vdash e \Rightarrow B$.

- **Case I-Var:** Given the conclusion $\Lambda; \Gamma \vdash \alpha_\mathcal{A}(b)$, we get to assume $\Gamma(b) = b$, and hence $\Lambda; \Gamma \vdash x \Rightarrow b$. By $I_1$, we have $b \subseteq P[x]$.
- **Case I-App:** Given the conclusion $\Lambda; \Gamma \vdash e_i \Rightarrow B_i$ for each $i$. By IH, we have $B_i \subseteq P[e_i]$ for each $i$. Then by Theorem A.3 we have $[c](B_i) \subseteq [c](P[e_i])$, and by $I_2$ we have $[c](P[e_i]) \subseteq P[e]$; hence $[c](B_i) \subseteq P[e]$. Applying $\alpha_\mathcal{A}$ to both sides, by Theorem A.10 we have $\alpha_\mathcal{A}([c](B_i)) \subseteq \alpha_\mathcal{A}(P[e])$. But $\alpha_\mathcal{A}(P[e]) = P[e]$ because range$(P) \subseteq \mathcal{A}$, hence we get $\alpha_\mathcal{A}([c](B_i)) \subseteq P[e]$ as required.

**Lemma A.15 (Untypeability).** Let $E = \lambda\overline{x_i}. e$, $t = \overline{b_i} \rightarrow b$, $e^* = r e$, where $\Lambda(r) = b \rightarrow b$. If $I_1 \land I_2 \land I_3$ and range$(P) \subseteq \mathcal{A}$, then $\Lambda; \cdot \nabla_\mathcal{A} E = t$.

**Proof.** Assume the contrary: that we can derive $\Lambda; \cdot \nabla_\mathcal{A} E \Leftarrow t$. Then we must be able to derive $\Lambda; \Gamma \vdash e \Leftarrow b$, where $\Gamma = [x_i: \overline{b_i}]$. By C-Base, if we derive $\Lambda; \Gamma \vdash e \Rightarrow B$, it must be that $b \subseteq B$ (note that inference result always exists and is unique).

By Theorem A.14, $\Lambda; \Gamma \vdash e^* \Rightarrow B^* \subseteq P[e^*]$, but by $I_3$ we have $P[e^*] = \bot$, hence $B^* = \bot$. By I-App and considering $\Lambda(r)$, we have $B^* = \sigma b$, where $\sigma = \text{mgu}(B, b)$. The only $\sigma$ such that $\sigma b = \bot$ is $\sigma_\bot$, hence we get $\text{mgu}(B, b) = \sigma_\bot$. But then it cannot be that $b \subseteq B$, a contradiction.

---

**Proc. ACM Program. Lang., Vol. 1, No. CONF, Article 1. Publication date: January 2018.**
B EVALUATION RESULTS

appBoth: \((a \rightarrow b) \rightarrow (a \rightarrow c) \rightarrow a \rightarrow (b, c)\)

**Demand Analysis**

\(((,) (arg2 arg0) (arg1 arg0))
\(((,) (\$) (arg2 arg0) (arg1 arg0))
\(((,) (arg2 (fromJust Nothing)) (arg1 arg0))
\(((,) (arg2 (head [])) (arg1 arg0))
\(((,) (arg2 (last [])) (arg1 arg0))

**No Demand Analysis**

\(((,) (arg2 arg0) (arg1 arg0))
\(((,) (\$) (arg2 arg0) (arg1 arg0))
\(((,) (arg2 (fromJust Nothing)) (arg1 arg0))
\(((,) (arg2 (head [])) (arg1 arg0))
\(((,) (arg2 (last [])) (arg1 arg0))

**No Relevancy**

\(((,) (arg2 arg0) (arg1 arg0))

**countFilter: \((a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow \text{Int}\)**

**Demand Analysis**

\(\text{length } (\text{map arg1 arg0})
\(\text{length } (\text{dropWhile arg1 arg0})
\(\text{length } (\text{filter arg1 arg0})
\(\text{length } (\text{takeWhile arg1 arg0})
\(\text{length } (\text{repeat } (\text{arg1 (head arg0)}))

**No Demand Analysis**

\(\text{length } (\text{map arg1 arg0})
\(\text{length } (\text{dropWhile arg1 arg0})
\(\text{length } (\text{filter arg1 arg0})
\(\text{length } (\text{takeWhile arg1 arg0})
\(\text{length } (\text{repeat } (\text{arg1 (head arg0)}))

**No Relevancy**

\(\text{length } (\text{map arg1 arg0})
\(\text{length } (\text{dropWhile arg1 arg0})
\(\text{length } (\text{filter arg1 arg0})
\(\text{length } (\text{takeWhile arg1 arg0})
\(\text{length } (\text{repeat arg1})

**mbElem: \(\text{Eq a} \Rightarrow a \rightarrow [a] \rightarrow \text{Maybe a}\)**

**Demand Analysis**

\(\text{lookup arg1 (zip arg0 [])}
\(\text{lookup arg1 (zip [] arg0)}
\(\text{lookup arg1 (zip arg0 arg0)}

No Demand Analysis

lookup arg1 (zip arg0 [])
lookup arg1 (zip [] arg0)
lookup arg1 (zip arg0 arg0)

No Relevancy

Just arg1
listToMaybe arg0
Nothing
listToMaybe (cycle arg0)
listToMaybe (init arg0)

hoogle01: (a → b) → [a] → b

Demand Analysis
arg1 (head arg0)
arg1 (last arg0)
($) arg1 (head arg0)
($) arg1 (last arg0)
arg1 (head (cycle arg0))

No Demand Analysis
arg1 (head arg0)
arg1 (last arg0)
($) arg1 (head arg0)
($) arg1 (last arg0)
arg1 (head (cycle arg0))

No Relevancy
arg1 (head arg0)
arg1 (last arg0)
arg1 (fromJust (listToMaybe arg0))
arg1 (fromJust Nothing)
($) arg1 (head arg0)

fromFirstMaybes: a → [Maybe a] → a

Demand Analysis
fromMaybe arg1 (head arg0)
fromMaybe arg1 (last arg0)
maybe arg1 id (head arg0)
maybe arg1 id (last arg0)
bool arg1 arg1 (null arg0)
No Demand Analysis

fromRight arg1 (Left arg0)
fromLeft arg1 (Right arg0)
fromMaybe arg1 (head arg0)
fromMaybe arg1 (last arg0)
fromLeft arg1 (Right (catMaybes arg0))

No Relevancy

fromLeft arg1 (Right arg0)
fromRight arg1 (Left arg0)
fromLeft arg1 (Right arg1)
fromRight arg1 (Right arg1)
fromLeft arg1 (Left arg1)

**mbAppFirst**: \( b \rightarrow (a \rightarrow b) \rightarrow [a] \rightarrow b \)

Demand Analysis

maybe arg2 arg1 (listToMaybe arg0)
fromMaybe arg2 (listToMaybe (map arg1 arg0))
fromLeft arg2 (Right (arg1 (head arg0)))

No Demand Analysis

maybe arg2 arg1 (listToMaybe arg0)
fromLeft (arg1 (head arg0)) (Right arg2)
fromRight (arg1 (head arg0)) (Right arg2)
fromLeft (arg1 (head arg0)) (Left arg2)
fromRight (arg1 (head arg0)) (Left arg2)

No Relevancy

arg1 (head arg0)
arg1 (last arg0)
fromLeft arg2 (Right arg1)
fromRight arg2 (Left arg1)
($) arg1 (head arg0)

**appendN**: \( \text{Int} \rightarrow [a] \rightarrow [a] \)

Demand Analysis

drop arg1 arg0
take arg1 arg0
drop arg1 (cycle arg0)
take arg1 (cycle arg0)
drop arg1 (init arg0)
No Demand Analysis

\[
\begin{align*}
drop & \text{arg1 arg0} \\
take & \text{arg1 arg0} \\
drop & \text{arg1 (cycle arg0)} \\
take & \text{arg1 (cycle arg0)} \\
drop & \text{arg1 (init arg0)}
\end{align*}
\]

No Relevancy

\[
\begin{align*}
cycle & \text{arg0} \\
init & \text{arg0} \\
reverse & \text{arg0} \\
tail & \text{arg0} \\
drop & \text{arg1 arg0}
\end{align*}
\]

\textbf{firstMaybe: } \texttt{[Maybe a] \rightarrow a} 

\[
\begin{align*}
\text{Demand Analysis} \\
& \text{fromJust (head arg0)} \\
& \text{fromJust (last arg0)} \\
& \text{head (catMaybes arg0)} \\
& \text{last (catMaybes arg0)} \\
& \text{fromJust (head (cycle arg0))}
\end{align*}
\]

No Demand Analysis

\[
\begin{align*}
& \text{fromJust (head arg0)} \\
& \text{fromJust (last arg0)} \\
& \text{head (catMaybes arg0)} \\
& \text{last (catMaybes arg0)} \\
& \text{fromJust (head (cycle arg0))}
\end{align*}
\]

No Relevancy

\[
\begin{align*}
& \text{head []} \\
& \text{last []} \\
& \text{fromJust (head arg0)} \\
& \text{fromJust (last arg0)} \\
& \text{fromJust Nothing}
\end{align*}
\]

\textbf{mapEither: } \texttt{(a \rightarrow Either b c) \rightarrow [a] \rightarrow ([b], [c])} 

\[
\begin{align*}
\text{Demand Analysis} \\
& \text{partitionEithers (map arg1 arg0)} \\
& \text{partitionEithers (repeat (arg1 (head arg0)))} \\
& \text{partitionEithers (repeat (arg1 (last arg0)))} \\
& \text{curry (last []) arg1 arg0}
\end{align*}
\]
No Demand Analysis
partitionEithers (map arg1 arg0)
partitionEithers (repeat (arg1 (head arg0)))
partitionEithers (repeat (arg1 (last arg0)))
curry (head []) arg1 arg0
curry (last []) arg1 arg0

No Relevancy
partitionEithers (map arg1 arg0)

head-rest: [a] → (a, [a])

Demand Analysis
fromJust (uncons arg0)
(,) (last arg0) arg0
(,) (head arg0) arg0
(,) (last arg0) []
(,) (head arg0) []

No Demand Analysis
fromJust (uncons arg0)
(,) (last arg0) arg0
(,) (head arg0) arg0
(,) (last arg0) []
(,) (head arg0) []

No Relevancy
head []
last []
fromJust (uncons arg0)
fromJust Nothing
(,) (head arg0) arg0

applyPair: (a → b, a) → b

Demand Analysis
uncurry ($) arg0
uncurry id arg0
($) (fst arg0) (snd arg0)

No Demand Analysis
uncurry ($) arg0
uncurry id arg0
($) (fst arg0) (snd arg0)
uncurry (head []) arg0
($) (head []) arg0
No Relevancy
uncurry id arg0
uncurry ($) arg0
fromJust Nothing
head []
last []

\textbf{maybe:} Maybe a \rightarrow a \rightarrow \text{Maybe} a

\textbf{Demand Analysis}
Just (fromMaybe arg0 arg1)
Just (maybe arg0 id arg1)
listToMaybe (repeat (fromMaybe arg0 arg1))
curry (last []) arg1 arg0
curry (last []) arg0 arg1

\textbf{No Demand Analysis}
Just (fromMaybe arg0 arg1)
fromRight arg1 (Left arg0)
fromLeft arg1 (Right arg0)
fromRight (Just arg0) (Left arg1)
fromLeft (Just arg0) (Left arg1)

No Relevancy
Just arg0
Nothing
fromRight arg1 (Left arg0)
fromLeft arg1 (Right arg0)
($) id arg1

\textbf{multiAppPair:} (a \rightarrow b, a \rightarrow c) \rightarrow a \rightarrow (b, c)

\textbf{Demand Analysis}
(,) (($) (fst arg1) arg0) (($) (snd arg1) arg0)
curry (last []) arg0 arg1

\textbf{No Demand Analysis}
(,) (($) (fst arg1) arg0) (($) (snd arg1) arg0)
curry (fromJust Nothing) arg0 arg1
curry (head []) arg0 arg1
curry (last []) arg0 arg1

No Relevancy
fromJust Nothing
head []
last []

\textbf{applyNtimes:} (a \rightarrow a) \rightarrow a \rightarrow \text{Int} \rightarrow a
**Demand Analysis**

\[
\begin{align*}
\text{arg2 } & (\text{repeat arg1}) \text{ arg0} \\
\text{arg2 } & (\text{head (replicate arg0 arg1)}) \\
\text{arg2 } & (\text{last (replicate arg0 arg1)}) \\
\text{head (replicate arg0 arg2 arg1)} \\
\text{last (replicate arg0 arg2 arg1)}
\end{align*}
\]

**No Demand Analysis**

\[
\begin{align*}
\text{arg2 } & (\text{fromRight arg1 (Left arg0)}) \\
\text{arg2 } & (\text{fromLeft arg1 (Right arg0)}) \\
\text{fromRight (arg2 arg1) (Left arg0)} \\
\text{fromLeft (arg2 arg1) (Right arg0)} \\
\text{arg2 } & (\text{repeat arg1) arg0})
\end{align*}
\]

**No Relevancy**

\[
\begin{align*}
\text{arg2 arg1} \\
\text{arg2 (arg2 arg1)} \\
\text{($) arg2 arg1} \\
\text{fromMaybe (arg2 arg1) (Just (arg2 arg1))} \\
\text{fromMaybe (arg2 arg1) Nothing}
\end{align*}
\]

**splitAtFirst: a → [a] → ([a], [a])**

**Demand Analysis**

\[
\begin{align*}
(,) & \text{ arg0 (repeat arg1)} \\
\text{swap (),( arg0 (repeat arg1))} \\
\text{splitAt (length arg0) (repeat arg1)} \\
(,) & \text{ (repeat arg1) (cycle arg0)} \\
(,) & \text{ (repeat arg1) (init arg0)}
\end{align*}
\]

**No Demand Analysis**

\[
\begin{align*}
(,) & \text{ arg0 (repeat arg1)} \\
\text{swap (),( arg0 (repeat arg1))} \\
\text{splitAt (length arg0) (repeat arg1)} \\
(,) & \text{ (repeat arg1) (cycle arg0)} \\
(,) & \text{ (repeat arg1) (init arg0)}
\end{align*}
\]

**No Relevancy**

\[
\begin{align*}
(,) & \text{ arg0 arg0} \\
(,) & \text{ (cycle arg0) (cycle arg0)} \\
(,) & \text{ (init arg0) (init arg0)} \\
(,) & \text{ (reverse arg0) (reverse arg0)} \\
(,) & \text{ (tail arg0) (tail arg0)}
\end{align*}
\]

**eitherTriple: Either a b → Either a b → Either a b**
Demand Analysis

```
bool arg1 arg1 (isRight arg0)
bool arg1 arg1 (isLeft arg0)
fromMaybe arg1 (listToMaybe (repeat arg0))
bool arg1 arg0 (last [])
bool arg1 arg0 (and [])
```

No Demand Analysis

```
fromLeft arg1 (Left arg0)
fromRight arg1 (Left arg0)
bool arg1 arg0 False
bool arg1 arg0 True
bool arg1 arg0 otherwise
```

No Relevancy

```
fromMaybe arg1 Nothing
bool arg1 arg1 False
bool arg1 arg1 True
bool arg1 arg1 otherwise
fromLeft arg1 (Right arg1)
```

\[\textbf{multiApp:} \ ((a \to b \to c) \to (a \to b)) \to a \to c \]

Demand Analysis

```
arg2 arg0 (arg1 arg0)
arg2 (fromJust Nothing) (arg1 arg0)
arg2 (head []) (arg1 arg0)
arg2 (last []) (arg1 arg0)
arg2 arg0 (arg1 (fromJust Nothing))
```

No Demand Analysis

```
arg2 arg0 (arg1 arg0)
arg2 arg0 ((\$) arg1 arg0)
arg2 (fromJust Nothing) (arg1 arg0)
arg2 (head []) (arg1 arg0)
arg2 (last []) (arg1 arg0)
```

No Relevancy

```
arg2 arg0 (arg1 arg0)
arg2 arg0 ((\$) arg1 arg0)
arg2 (snd ((\,) (arg1 arg0) arg0)) (fst ((\,) (arg1 arg0) arg0))
uncurry arg2 ((\,) arg0 (arg1 arg0))
arg2 (snd ((\,) arg0 arg0)) (arg1 (fst ((\,) arg0 arg0)))
```

\[\textbf{mergeEither:} \textit{Either a (Either a b)} \to \textit{Either a b} \]
Demand Analysis
fromRight (fromJust Nothing) arg0
fromRight (head []) arg0
fromRight (last []) arg0

No Demand Analysis
fromRight (fromJust Nothing) arg0
fromRight (head []) arg0
fromRight (last []) arg0

No Relevancy
fromJust Nothing
head []
last []
fromRight (fromJust Nothing) arg0
fromRight (head []) arg0

\textbf{firstRight:} \([\text{Either } a \ b] \rightarrow \text{Either } a \ b\)

Demand Analysis
head arg0
last arg0
head (cycle arg0)
last (cycle arg0)
head (init arg0)

No Demand Analysis
head arg0
last arg0
head (cycle arg0)
last (cycle arg0)
head (init arg0)

No Relevancy
head arg0
last arg0
head (cycle arg0)
last (cycle arg0)
head (init arg0)

\textbf{flatten:} \([[[a]]] \rightarrow [a]\)

Demand Analysis
head (head arg0)
last (head arg0)
concat (head arg0)
head (last arg0)
last (last arg0)
No Demand Analysis
head (head arg0)
last (head arg0)
concat (head arg0)
head (last arg0)
last (last arg0)

No Relevancy
[]
lefts []
rights []
catMaybes []
cycle []

pipe: [(a → a)] → (a → a)

Demand Analysis
foldr ($) arg0 arg1
($) (head arg1) arg0
($) (last arg1) arg0
foldr id arg0 arg1
foldr id arg0 (cycle arg1)

No Demand Analysis
foldr ($) arg0 arg1
fromLeft arg0 (Right arg1)
($) (head arg1) arg0
($) (last arg1) arg0
foldr id arg0 arg1

No Relevancy
fromLeft arg0 (Right arg0)
fromRight arg0 (Right arg0)
fromLeft arg0 (Left arg0)
fromRight arg0 (Left arg0)
fromMaybe arg0 (Just arg0)

firstKey: [(a,b)] → a

Demand Analysis
(!) [] (length arg0)
($) (last []) arg0

No Demand Analysis
(!) [] (length arg0)
($) (last []) arg0
($) (head []) arg0
($) (fromJust Nothing) arg0
splitStr: String → Char → [String]

No Demand Analysis
repeat (showChar arg0 arg1)
repeat ((:) arg0 arg1)
(:) (repeat arg0) (repeat arg1)
repeat ((++) arg1 (repeat arg0))
repeat (showString arg1 (repeat arg0))

No Relevancy
repeat arg1
[]
repeat (cycle arg1)
repeat (init arg1)
repeat (reverse arg1)

areEq: Eq a ⇒ a → a → Maybe a

Demand Analysis
fromMaybe (Just arg1) (lookup arg0 [])
Just (fromMaybe arg1 (lookup arg0 []))

No Demand Analysis
fromMaybe (Just arg1) (lookup arg0 [])
Just (fromMaybe arg1 (lookup arg0 []))
fromLeft (Just arg1) (Right ((,) arg0))
fromRight (Just arg1) (Left ((,) arg0))
fromLeft (Just arg1) (Right ((,) arg0 ))

No Relevancy
Just arg1
Nothing
listToMaybe (repeat arg1)
lookup arg1 []
fromJust Nothing

lookup: Eq a ⇒ [(a,b)] → a → b
 Demand Analysis
fromJust (lookup arg0 arg1)
head (maybeToList (lookup arg0 arg1))
last (maybeToList (lookup arg0 arg1))
fromJust (lookup arg0 (cycle arg1))
fromJust (lookup arg0 (init arg1))

No Demand Analysis
fromJust (lookup arg0 arg1)
head (maybeToList (lookup arg0 arg1))
last (maybeToList (lookup arg0 arg1))

No Relevancy
fromJust (lookup arg0 arg1)
fromJust Nothing
head []
last []

map: (a → b) → [a] → [b]

Demand Analysis
map arg1 arg0
repeat (arg1 (last arg0))
map arg1 (cycle arg0)
map arg1 (init arg0)
map arg1 (reverse arg0)

No Demand Analysis
map arg1 arg0
repeat (arg1 (last arg0))
map arg1 (cycle arg0)
map arg1 (init arg0)
map arg1 (reverse arg0)

No Relevancy
map arg1 arg0
lefts []
rights []
catMaybes []
concat []

resolveEither: Either a b → (a→b) → b

Demand Analysis
either arg0 id arg1
arg0 (head (lefts (repeat arg1)))
arg0 (last (lefts (repeat arg1)))
either arg0 (head []) arg1
either arg0 (last []) arg1
No Demand Analysis

`either arg0 id arg1`
`either arg0 (fromJust Nothing) arg1`

No Relevancy

`either arg0 id arg1`
`arg0 (fromJust Nothing)`
`arg0 (head [])`
`arg0 (last [])`
`fromRight (arg0 (fromJust Nothing)) arg1`

**firstMatch: [a] → (a → Bool) → a**

Demand Analysis

`last (dropWhile arg0 arg1)`
`head (dropWhile arg0 arg1)`
`last (filter arg0 arg1)`
`head (filter arg0 arg1)`
`last (takeWhile arg0 arg1)`

No Demand Analysis

`last (dropWhile arg0 arg1)`
`head (dropWhile arg0 arg1)`
`last (filter arg0 arg1)`
`head (filter arg0 arg1)`
`last (takeWhile arg0 arg1)`

No Relevancy

`head (dropWhile arg0 arg1)`
`last (dropWhile arg0 arg1)`
`head (filter arg0 arg1)`
`last (filter arg0 arg1)`
`head (takeWhile arg0 arg1)`

**test: Bool → a → Maybe a**

Demand Analysis

`bool (Just arg0) Nothing arg1`
`bool Nothing (Just arg0) arg1`
`bool (Just arg0) (Just arg0) arg1`
`Just (bool arg0 arg0 arg1)`
`curry (last []) arg0 arg1`

No Demand Analysis

`bool (Just arg0) Nothing arg1`
`bool Nothing (Just arg0) arg1`
`Just (fromLeft arg0 (Right arg1))`
`Just (fromRight arg0 (Left arg1))`
`bool (Just arg0) (Just arg0) arg1`
No Relevancy
Just arg0
Nothing
Just (bool arg0 arg0 arg1)
listToMaybe (repeat arg0)
listToMaybe []

intToBS: Int64 → ByteString

Demand Analysis
drop arg0 empty
take arg0 empty
toLazyByteString (int64Dec arg0)
toLazyByteString (int64HexFixed arg0)
toLazyByteString (int64LE arg0)

No Demand Analysis
drop arg0 empty
take arg0 empty
toLazyByteString (int64Dec arg0)
toLazyByteString (int64HexFixed arg0)
toLazyByteString (int64LE arg0)

repl-funs: (a → b) → Int → [a → b]

Demand Analysis
replicate arg0 arg1
cycle (replicate arg0 arg1)
init (replicate arg0 arg1)
reverse (replicate arg0 arg1)
tail (replicate arg0 arg1)

No Demand Analysis
replicate arg0 arg1
cycle (replicate arg0 arg1)
init (replicate arg0 arg1)
reverse (replicate arg0 arg1)
tail (replicate arg0 arg1)
No Relevancy
repeat arg1
replicate arg0 arg1
lefts []
rights []
catMaybes []

mapMaybes: (a → Maybe b) → [a] → Maybe b
Demand Analysis
arg1 (head arg0)
arg1 (last arg0)
($) arg1 (head arg0)
($) arg1 (last arg0)
arg1 (fromJust (listToMaybe arg0))

No Demand Analysis
arg1 (head arg0)
arg1 (last arg0)
($) arg1 (head arg0)
($) arg1 (last arg0)
arg1 (fromJust (listToMaybe arg0))

No Relevancy
arg1 (head arg0)
arg1 (last arg0)
arg1 (!!) arg0 (length arg0))
arg1 (head [])
arg1 (last [])

takeNdropM: Int → Int → [a] → ([a], [a])
Demand Analysis
splitAt arg2 (drop arg1 arg0)
splitAt arg2 (take arg1 arg0)
splitAt arg2 (take arg1 (cycle arg0))
splitAt arg2 (drop arg1 (cycle arg0))
splitAt arg2 (take arg1 (init arg0))

No Demand Analysis
splitAt arg2 (drop arg1 arg0)
splitAt arg2 (take arg1 arg0)
splitAt arg2 (take arg1 (cycle arg0))
splitAt arg2 (drop arg1 (cycle arg0))
splitAt arg2 (take arg1 (init arg0))
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>cartProduct</code></td>
<td>Takes two lists and returns a list of tuples, where each tuple contains an element from the first list and an element from the second list.</td>
</tr>
<tr>
<td><code>Demand Analysis</code></td>
<td>Performs operations on the input lists.</td>
</tr>
<tr>
<td><code>No Demand Analysis</code></td>
<td>Performs no operations on the input lists.</td>
</tr>
</tbody>
</table>

### Code Snippets

#### `cartProduct` Function
```haskell
cartProduct: [a] → [b] → [(a,b)]
```

#### Demand Analysis
```haskell
repeat (zip arg1 arg0)
repeat (cycle (zip arg1 arg0))
repeat (init (zip arg1 arg0))
repeat (reverse (zip arg1 arg0))
repeat (tail (zip arg1 arg0))
```

#### No Demand Analysis
```haskell
repeat (zip arg1 arg0)
repeat (cycle (zip arg1 arg0))
repeat (init (zip arg1 arg0))
repeat (reverse (zip arg1 arg0))
repeat (tail (zip arg1 arg0))
```

#### Hoogle02 Function
```haskell
hoogle02: b → (a → b) → [a] → b
```

#### Demand Analysis
```haskell
maybe arg2 arg1 (listToMaybe arg0)
fromMaybe arg2 (listToMaybe (map arg1 arg0))
```

#### No Demand Analysis
```haskell
maybe arg2 arg1 (listToMaybe arg0)
fromLeft (arg1 (head arg0)) (Right arg2)
fromRight (arg1 (head arg0)) (Left arg2)
```

#### No Relevancy
```haskell
arg1 (head arg0)
arg1 (last arg0)
```
containsEdge: \([\text{Int}] \rightarrow (\text{Int,Int}) \rightarrow \text{Bool}\)

<table>
<thead>
<tr>
<th>Demand Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{null (repeat (,, arg0 arg1))})</td>
</tr>
<tr>
<td>(\text{null (replicate (length arg1) arg0)})</td>
</tr>
<tr>
<td>(\text{null (replicate (head arg1) arg0)})</td>
</tr>
<tr>
<td>(\text{null (replicate (last arg1) arg0)})</td>
</tr>
<tr>
<td>(\text{null (repeat (,, arg1 arg0))})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No Demand Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{null (fromLeft arg1 (Right arg0))})</td>
</tr>
<tr>
<td>(\text{null (fromRight arg1 (Left arg0))})</td>
</tr>
<tr>
<td>(\text{isLeft (Right (,, arg0 arg1))})</td>
</tr>
<tr>
<td>(\text{isRight (Right (,, arg0 arg1))})</td>
</tr>
<tr>
<td>(\text{isLeft (Left (,, arg0 arg1))})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No Relevancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{False})</td>
</tr>
<tr>
<td>(\text{True})</td>
</tr>
<tr>
<td>(\text{otherwise})</td>
</tr>
<tr>
<td>(\text{null arg1})</td>
</tr>
<tr>
<td>(\text{isJust Nothing})</td>
</tr>
</tbody>
</table>

**app3:** \((a \rightarrow b \rightarrow c \rightarrow d) \rightarrow a \rightarrow c \rightarrow b \rightarrow d\)

<table>
<thead>
<tr>
<th>Demand Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{arg3 arg2 arg0 arg1})</td>
</tr>
<tr>
<td>(\text{fromMaybe (arg3 arg2 arg0 arg1) Nothing})</td>
</tr>
<tr>
<td>(\text{arg3 (fst ((,, arg2 arg0)) (snd ((,, arg2 arg0)) arg1)})</td>
</tr>
<tr>
<td>(\text{arg3 (snd ((,, arg0 arg2)) (fst ((,, arg0 arg2)) arg1)})</td>
</tr>
<tr>
<td>(\text{arg3 arg2 arg0 (fromMaybe arg1 Nothing)})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No Demand Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{arg3 arg2 arg0 arg1})</td>
</tr>
<tr>
<td>(\text{fromMaybe (arg3 arg2 arg0 arg1) Nothing})</td>
</tr>
<tr>
<td>(\text{arg3 (fst ((,, arg2 arg0)) (snd ((,, arg2 arg0)) arg1)})</td>
</tr>
<tr>
<td>(\text{arg3 (snd ((,, arg0 arg2)) (fst ((,, arg0 arg2)) arg1)})</td>
</tr>
<tr>
<td>(\text{arg3 arg2 arg0 (fromMaybe arg1 Nothing)})</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No Relevancy</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\text{arg3 arg2 arg0 arg1})</td>
</tr>
<tr>
<td>(\text{fromMaybe (arg3 arg2 arg0 arg1) Nothing})</td>
</tr>
<tr>
<td>(\text{fromLeft (arg3 arg2 arg0 arg1) (Right arg2)})</td>
</tr>
<tr>
<td>(\text{fromRight (arg3 arg2 arg0 arg1) (Left arg2)})</td>
</tr>
<tr>
<td>(\text{fromLeft (arg3 arg2 arg0 arg1) (Right arg1)})</td>
</tr>
</tbody>
</table>

**indexesOf:** \(((a,\text{Int})) \rightarrow ((a,\text{Int})) \rightarrow [a] \rightarrow \text{[Int]} \rightarrow \text{[Int]}\)
No Demand Analysis
\[
\text{fromLeft arg0 (Right (., arg1 arg2))} \\
\text{fromRight arg0 (Left (., arg1 arg2))} \\
\text{fromLeft arg0 (Right (., arg2 arg1))} \\
\text{fromRight arg0 (Left (., arg2 arg1))}
\]

No Relevancy
\[
\text{lefts []} \\
\text{rights []} \\
\text{catMaybes []} \\
\text{concat []} \\
\text{cycle []}
\]

**both**: \((a \rightarrow b) \rightarrow (a, a) \rightarrow (b, b)\)

Demand Analysis
\[
(.,) (arg1 (snd arg0)) (arg1 (fst arg0))
\]

No Demand Analysis
\[
(.,) (arg1 (snd arg0)) (arg1 (fst arg0))
\]

No Relevancy
\[
\text{head []} \\
\text{last []} \\
\text{head (maybeToList Nothing)} \\
\text{last (maybeToList Nothing)} \\
\text{head (lefts [])}
\]

**zipWithResult**: \((a \rightarrow b) \rightarrow [a] \rightarrow [(a, b)]\)

Demand Analysis
\[
\text{zip arg0 (map arg1 [])} \\
\text{zip arg0 (map arg1 arg0)}
\]

No Demand Analysis
\[
\text{zip arg0 (map arg1 [])} \\
\text{zip arg0 (map arg1 arg0)}
\]

No Relevancy
\[
\text{lefts []} \\
\text{rights []} \\
\text{catMaybes []} \\
\text{concat []} \\
\text{cycle []}
\]

**rights**: \([\text{Either a b}] \rightarrow \text{Either a [b]}\)

Demand Analysis
\[
\text{Right (rights arg0)} \\
(!!) [] (\text{length arg0})
\]
No Demand Analysis
\[
\text{Right (rights arg0)}
\]
\[
[\text{[![]]} \text{[]} \text{(length arg0)}]
\]

No Relevancy
\[
\text{head []}
\]
\[
\text{last []}
\]
\[
\text{Right []}
\]
\[
\text{fromJust Nothing}
\]
\[
\text{Right (rights arg0)}
\]

**mbToEither**: \(\text{Maybe a} \rightarrow \text{b} \rightarrow \text{Either a b}\)

Demand Analysis
\[
\text{curry (last []) arg1 arg0}
\]
\[
\text{curry (last []) arg0 arg1}
\]

No Demand Analysis
\[
\text{fromRight (Right arg0) (Left arg1)}
\]
\[
\text{fromLeft (Right arg0) (Right arg1)}
\]
\[
\text{Right (fromRight arg0 (Left arg1))}
\]
\[
\text{Right (fromLeft arg0 (Right arg1))}
\]
\[
\text{curry (fromJust Nothing) arg1 arg0}
\]

No Relevancy
\[
\text{Right arg0}
\]
\[
\text{fromJust Nothing}
\]
\[
\text{Left (fromJust arg1)}
\]
\[
\text{head []}
\]
\[
\text{last []}
\]

**singleList**: \(\text{Int} \rightarrow [\text{Int}]\)

Demand Analysis
\[
\text{repeat arg0}
\]
\[
\text{cycle (repeat arg0)}
\]
\[
\text{init (repeat arg0)}
\]
\[
\text{reverse (repeat arg0)}
\]
\[
\text{tail (repeat arg0)}
\]

No Demand Analysis
\[
\text{repeat arg0}
\]
\[
\text{cycle (repeat arg0)}
\]
\[
\text{init (repeat arg0)}
\]
\[
\text{reverse (repeat arg0)}
\]
\[
\text{tail (repeat arg0)}
\]
No Relevancy
replicate arg0 arg0
repeat arg0
[]
(·) arg0 []
iterate id arg0

head-tail: [a] → (a,a)

Demand Analysis
(,) (last arg0) (last arg0)
(,) (head arg0) (head arg0)
last (zip [] arg0)
head (zip [] arg0)
($) (last []) arg0

No Demand Analysis
(,) (last arg0) (last arg0)
(,) (head arg0) (head arg0)
last (zip [] arg0)
head (zip [] arg0)
($) (last []) arg0

No Relevancy
fromJust Nothing
head []
last []
head (zip arg0 arg0)
lst (zip arg0 arg0)

2partApp: (a → b) → (b → c) → [a] → [c]

Demand Analysis
map arg1 (map arg2 arg0)
repeat (arg1 (arg2 (last arg0)))
repeat (arg1 (arg2 (head arg0)))
iterate id (arg1 (arg2 (last arg0)))
iterate' id (arg1 (arg2 (last arg0)))

No Demand Analysis
map arg1 (map arg2 arg0)
repeat (arg1 (arg2 (last arg0)))
repeat (arg1 (arg2 (head arg0)))
repeat (arg1 (($) arg2 (last arg0)))
repeat (arg1 (($) arg2 (head arg0)))
No Relevancy

fromJust (fromJust Nothing)
head (fromJust Nothing)
last (fromJust Nothing)