Speeding up Type-Driven Program Synthesis with Polymorphic Succinct Types

ZHENG GUO, University of California, San Diego, USA

1 INTRODUCTION

Several component-based program synthesis techniques [1, 4] rely on deductive reasoning to reduce the search space. While effective for many interesting programs, these techniques scale poorly with the number of given components and the depth of the program (i.e. the height of its AST). In this project, we explore an approach to improving the scalability of type-driven synthesis [4] by ruling out useless components, whose argument types are known to be uninhabited. Unfortunately, type inhabitation is undecidable in the presence of polymorphism, because the set of reachable types in a finite type environment can be infinite. To address this problem, we abstract the actual, fine-grained types of components into coarse-grained *succinct types*, a concept we borrow from InSynth [2].

We extend succinct types to support both datatypes and polymorphism, which requires a novel notion of unification between succinct types. Then, we can compute the over-approximation of type reachability, exclude components with unreachable argument types and build the *type transfer graph* (TTG), which summarizes how to construct a term of a given type from terms of other types [2]. We apply this abstraction to Synquid [4], a state-of-the-art type-driven program synthesis tool. The evaluation results indicate that our approach efficiently prevents enumeration of useless components in Synquid's bottom-up term exploration phase [4].

2 POLYMORPHIC SUCCINCT TYPES

Succinct types were proposed in InSynth [2] as an abstraction over simple types, which ignores the order of function arguments and removes the duplicates. This idea is generalized to refinement types in Synquid [4] by abstracting all refinements to `true`.[1] For example, consider the following component

$$\texttt{eq} :: \texttt{x} : \texttt{Int} \rightarrow \texttt{y} : \texttt{Int} \rightarrow \{\texttt{Bool} | \nu == (\texttt{x} == \texttt{y})\}$$

has the succinct representation

$$\texttt{eq}::\{\texttt{Int}\} \rightarrow \texttt{Bool}$$

which indicates that `eq` enables us to construct a `Bool` term, as long as we can construct an `Int` term. To generalize succinct types to ML-style polymorphism, we extend their syntax as follows:

$$\begin{array}{lll} t_\alpha & ::= & \forall\{\alpha_1, \alpha_2, \ldots, \alpha_k\}.t_s \quad \text{type schema} \\ t_s & ::= & \{t_s, t_s, \ldots, t_s\} \rightarrow t_r \quad \text{function} \\ t_d & ::= & DT \diamond \{t_s, t_s, \ldots, t_s\} \quad \text{datatype} \\ t_r & ::= & Bool \mid Int \mid \alpha \mid t_d \quad \text{function return type} \end{array}$$

For polymorphic types, we abstract over the order of bound type variables, similarly to how the original succinct types ignore the order of function arguments. And we design succinct datatypes as the composition of two parts: the outermost datatype and other types. Consider the following component

$$\texttt{delete} :: \texttt{t} : \texttt{BST}\ \alpha \rightarrow \texttt{x} : \alpha \rightarrow \{\texttt{BST}\ \alpha | \texttt{elems}\ \nu == \texttt{elems}\ \texttt{t} - [\texttt{x}]\}$$

[1] We leave the abstraction of refinements into coarse-grained representations to our future work.

Author's address: Zheng Guo, University of California, San Diego, CA, 92092, USA, zhg069@ucsd.edu.

This component contains both polymorphism and datatypes. We represent it concisely by extracting all the type variables and decomposing the datatypes as follows:

$$\texttt{delete} :: \forall\{\alpha\}.\ \{\alpha, \texttt{BST}\ \diamond\ \{\alpha\}\} \rightarrow \texttt{BST}\ \diamond\ \{\alpha\}$$

This definition enables us to abstract infinite sets of types into finite sets of succinct types. For example, with the polymorphic datatype `List` α and a ground type `Int`, we are able to construct infinitely many types: `List (List Int)`, `List (List (List Int))`, `List (List (List (List Int)))`, etc. However, in our representation all these types have the same abstraction as `List` $\diamond$ `{List, Int}`.

Every ML-style type τ_λ can be converted to a succinct type t_α using a conversion function $\sigma : \tau_\lambda \rightarrow t_s$, which is defined as follows:

$\sigma(\text{Bool}) = \text{Bool}, \sigma(\text{Int}) = \text{Int}$	primitive
$\sigma(\forall\alpha_1 \ldots \forall\alpha_n.\tau_\alpha) = \forall\{\alpha_1 \ldots \alpha_n\}.\ \sigma(\tau_\alpha)$	type schema
$\sigma(\tau_1 \rightarrow \tau_2) = \{\sigma(\tau_1)\} \cup Arg(\sigma(\tau_2)) \rightarrow Ret(\sigma(\tau_2))$	function
$\sigma(D) = D \diamond \emptyset$	datatypes without type variables
$\sigma(\tau_{DT}\ \tau) = Out(\tau_{DT}) \diamond (Out(\sigma(\tau)) \cup Tys(\sigma(\tau)) \cup Tys(\sigma(\tau_{DT})))$	datatypes with type variables

where

$$Arg(\{t_1, t_2, \ldots, t_n\} \rightarrow t_r) = \{t_1, t_2, \ldots, t_n\} \qquad Ret(\{t_1, t_2, \ldots, t_n\} \rightarrow t_r) = t_r$$
$$Out(outerDT \diamond Ts) = outerDT \qquad Tys(outerDT \diamond Ts) = Ts$$

3 CASE STUDY: PRUNING COMPONENTS IN SYNQUID

We study this type system by applying it to the bottom-up term exploration phase in Synquid.

Constructing TTG. With the abstraction of provided components as succinct types, the first phase of bottom-up term exploration in Synquid is to calculate the set of types which are reachable to our synthesis goal type. For example, if our synthesis goal is the function `delete` above, the user provides us these components:

```
Nil::List α
Cons::x:α → xs:List α → List α
add::x:Nat → y:Nat → {Int|ν == x + y}
toBST::xs:List α → s:Nat → {BST α|elems xs == telems ν && size ν == s}
```

We may construct the TTG as Figure 1. Each edge from `T1` to `T2` in this graph indicates that to construct a term with succinct type `T2` we need a term with type `T1`. And there is an edge from a source node to `T` if there is a variable with type `T` in the current scope. We say a type is reachable if there exists a path from any source node to the goal type containing this type in the graph. By evaluating the reachability, we are able to rule out components containing unreachable types. In this case, the type `Int, Bool, List`$\diamond$ $\{\alpha\}$ turns out to be unreachable, so the edges named `toBST, eq, Cons, Nil` would be pruned from the graph and therefore these components would be excluded from the following term exploration.

To formalize this idea, we propose a unification algorithm between succinct types. Let $S \underset{\Gamma}{\rightsquigarrow} T$ denote type T is reachable from type S and the reachability rule in InSynth [2] is modified as follows:

$$\textsc{Match}\ \frac{T \underset{\Gamma}{\rightsquigarrow} ? \qquad \text{unify}(T, R) = \delta \qquad S \rightarrow R \in \Gamma \qquad Arg(R) = \emptyset}{\delta S \underset{\Gamma}{\rightsquigarrow} T}$$

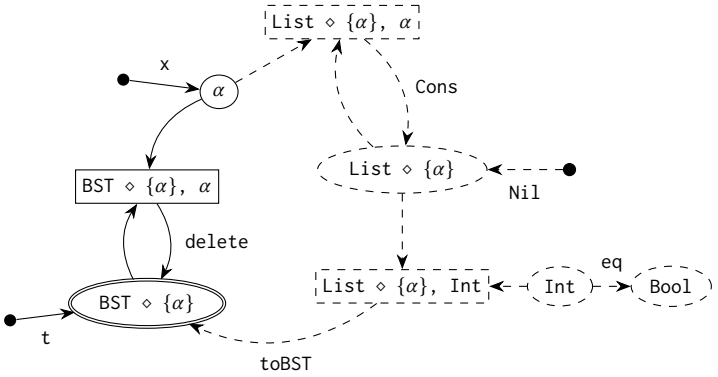

Fig. 1. Type transfer graph example. Double-lined ellipse nodes for the goal type, rectangle nodes for set of succinct types, single-lined ellipse nodes for single succinct type and solid dots for variables in scope. Dashed lines for pruned edges in the graph.

According to this rule, if we have a succinct component type $S \longrightarrow R$, and the type T and R can be unified under substitution δ, type T is reachable from the substituted type set δS. Then an edge from δS to T is added to the TTG.

Using TTG. Once the TTG is constructed, we perform term exploration by walking the graph, starting from the node that abstracts the goal type. We employ two further optimizations to speed up the search. First, we use the A* algorithm to reach complete terms faster [3]. Our second optimization is concerned with *condition abduction* [4]: after one branch of a conditional has been successfully synthesized, unlike Synquid, we resume the exploration of the other branch from where the first one left off, instead of restarting from scratch.

4 EVALUATION

The original Synquid benchmarks [4] use a minimal set of components. To evaluate the scalability of Synquid as the number of components increases, as well as the effect of our technique, we developed two additional benchmarks, which introduce useless components, and then increase the depth of the program to be synthesized or the number of components. As shown in Figure 2, with the solution depth increasing, the number of component combinations grow exponentially, and therefore running time of pure term exploration in Synquid grows drastically while our method takes almost the same time for all the depths.

On the other hand, Figure 3 confirms the benefits of our method in search space reduction, where the increase in number of components makes less difference in our method than that in Synquid. This result also shows that our method takes extra time to build the graph before term exploration, but the overhead is small in our benchmarks, although it would be more significant as the number of components increases. We also add extra components to other benchmarks of Synquid and the result is shown in Table 1. For most of the benchmarks with extra components, both the baseline Synquid and our method waste time trying some of the useless but reachable components. Notice that there are still some benchmarks such as list partition and BST delete that work better that the baseline, and this is likely due to resuming the search when switching between branches. To conclude, our method performs similarly to Synquid on simple benchmarks and significantly better on benchmarks with many useless component combinations.

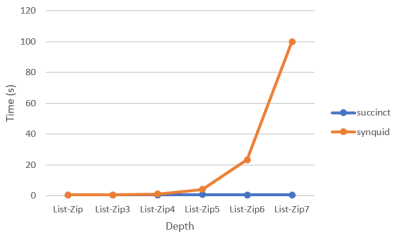

Fig. 2. Evaluation on depth benchmarks

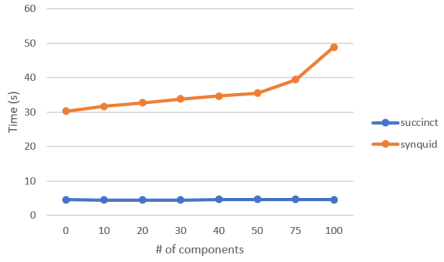

Fig. 3. Evaluation on component number benchmarks

Description	T_{min}(s)	T_{st}(s)	T_{sq}(s)
Make address book	1.36	**8.39**	10.14
Insert into binary heap	1.3	**1.21**	1.83
Delete node from BST	9.42	**4.46**	13.08
Merge two sorted lists	4.41	**3.83**	9.21
Delete value from a list	0.71	**0.81**	0.87
Is list empty	0.61	0.69	**0.67**
List partition	17.57	**12.60**	63.48
List replicate	0.51	**0.76**	0.99
Duplicate each element in a list	0.61	**0.80**	0.94

Table 1. Running time on SYNQUID benchmarks with minimal set of components in baseline SYNQUID(T_{min}), with extra components in our method(T_{st}) and with extra components in baseline SYNQUID(T_{sq}).

REFERENCES

[1] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 422–436. https://doi.org/10.1145/3062341.3062351

[2] Tihomir Gvero, Viktor Kuncak, Ivan Kuraj, and Ruzica Piskac. 2013. Complete Completion Using Types and Weights. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '13)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/2491956.2462192

[3] Woosuk Lee, Kihong Heo, Rajeev Alur, and Mayur Naik. 2018. Accelerating Search-based Program Synthesis Using Learned Probabilistic Models. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018)*. ACM, New York, NY, USA, 436–449. https://doi.org/10.1145/3192366.3192410

[4] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 522–538. https://doi.org/10.1145/2908080.2908093