An Essay on Program Synthesis

Advanced Topics in Programming Languages

The undecidability of program synthesis inherently limits its use to narrow problem domains. Therefore, rather than aiming to synthesize full, complex programs, we should use program synthesis as a means of augmenting human-written programs.

In *'Program Synthesis from Polymorphic Refinement Types'*, Polikarpova, Kuraj and Solar-Lezama design and implement Synquid, a domain specific language that synthesizes ML-style functional programs from a type specification. While Synquid is able to generate concise solutions to a variety of benchmarks, its inability to scale with complexity leaves its practical usage uncertain.

Yoon, Lee and Yi develop in *'Inductive Program Synthesis via Iterative Forward-Backward Abstract Interpretation'* Simba, a program synthesis tool that generates a program given a set of input-output pairs, using abstract interpretation to prune the solution search space. Simba excels in its intended domain of bit-vector functions, but the complexity of its system similarly makes its application to more complex domains untenable.

Testing the limits of Synquid

Polikarpova, et al.'s Synquid generates functional programs based on a refinement type signature, where a refinement type is a data type annotated with invariants we want to satisfy. For example:

$$\texttt{replicate} :: n : \texttt{Nat} \to x : \alpha \to \{\nu : \texttt{List}\ \alpha \mid \texttt{len}\ \nu \geq n\}$$

From the refined return type $\{\nu : \texttt{List}\ \alpha \mid \texttt{len}\ \nu \geq n\}$, we know that the output list must contain n elements of type α. But since x is the only element of type α in the context of `replicate`, Synquid deduces that the function must return a list xs as it has no other way of constructing values of type α:

```
replicate = λn. λx. if n ≤ 0
  then Nil
  else Cons x (replicate (dec n) x)
```

In essence, Synquid generates terms as a system of solving constraints over a type system. From a specification type with refinements, Synquid searches for a term t whose type and refinement is a subtype to that of the specification. This subtyping constraint then generates a predicate P that must be true for t to be valid given the specification. For example, when synthesizing `replicate`, Synquid tries $t := \texttt{Nil}$, generating the predicate $P := n \leq 0$ that must be true for $\texttt{len}\ \nu \geq n$. If P is trivially true, then t is the generated term that matches our specified type. Otherwise, we generate a term $\bar{t}$ that fits the specified type assuming $\neg P$ and output $\texttt{if}\ P\ \texttt{then}\ t\ \texttt{else}\ \bar{t}$. At the core we see that the terms Synquid synthesizes are dictated by the constraints defined for a given refined type.

Unfortunately, Synquid requires that any refined type be scalar. This means that terms that require auxiliary functions cannot be synthesized from the get-go. This restriction is notable as the target domain of Synquid is recursive functional programs for which auxiliary functions are essential in capturing the behavior that would otherwise be implemented using nested loops in imperative style. This therefore poses a great limitation in the programs that we would want to naturally specify, requiring that all auxiliary functions be given external implementations.

Even when decomposing a program to this granular level, the structure imposed on the type refinements results in Synquid being unable to synthesize certain problems. Polikarpova, et al. mention how they were only able to express 12 out of 25 complex benchmarks using Synquid. In particular, they fail to implement filtering and the manipulation of nested structures. This implies that Synquid is good

at creating data structures that preserve complex invariants, but is unable to modify such structures. Specifically, the restrictions that allow Synquid to synthesize a correct term, produce friction once we start to operate on them.

Thus, despite being targeted toward recursive functional programs, Synquid's limitations make recursive behavior either difficult to express, or outright unfeasible.

Synthesizing by example

The tool Simba that Yoon et al. describe, uses abstract interpretation to synthesize a program as defined by a set of input-output pairs. Given a problem domain (e.g. bit-vectors) and a grammar of expressions that operates over this domain (e.g. bitwise operators), Simba must also be given an abstract domain that will be used to prune the program search space.

Yoon, et al. state that Simba is guaranteed to find a solution if it exists. We may be tempted to then believe that this approach can synthesize a solution for all problems. However, since we are using input-output pairs as our specification, the resulting program must extrapolate to any pair not given. This means that we could have an edge case not covered in our specification that does not fit with the generated program. Thus, the actual correctness of the solution is dependent on whether the provided examples fully cover the problem domain.

Additionally, this guarantee of finding a solution assumes infinite time of execution. When given a large number of examples, Simba may find it difficult to generate solution that matches all of them, potentially leading to a time out. For the 1875 benchmarks that Yoon, et al. evaluate, 159 of these result in Simba timing out over a limit of one hour.

Finally, we must address the elephant in the room, namely that a suitable abstract domain is needed for Simba to work in the first place. Note that this abstract domain must be specific enough to cover the entire grammar of programs possible, requiring both forward and backwards abstract operators for every expression in the grammar. Even for the relatively simple domain of bit-vector expressions, Yoon, et al. combine three abstract domains to be able to capture the full breadth of behavior. This requirement limits Simba to problems with linear execution paths. More complex control flow will require the approximation of a fixed point to not time-out, which further leads to a loss of precision that prevents Simba identifying a solution.

So, like Synquid, Simba is also limited to synthesizing a small set of problems specific to a narrow domain. While it is very successful at this task, Simba's requirements relegate it to playing a complementary role in programming for efficiently generating programs that would otherwise require a brute-force approach to construct.

Hands-on Synthesis

Even when program synthesis works, should we prefer it over writing programs ourselves?

When it comes to Synquid, it is clear that it fails as a standalone programming paradigm as the mapping from refinement types to programs is largely unintuitive. Even with the simple "*hello world!*" example from earlier, it is not immediately obvious from the specification type that `replicate` must output a list of xs. In fact, we could extend `replicate`'s type to the following, adding a second argument y of type α:

$$n : \texttt{Nat} \to x : \alpha \to y : \alpha \to \{\nu : \texttt{List}\ \alpha \mid \texttt{len}\ \nu \geq n\}$$

The resulting function that Synquid generates is the same, modulo the extra argument. This shows a disconnect between user intention — that the output should contain both x and y – and program synthesis which exclusively uses x. In fact, one could argue that Synquid is a flavor of logic program-

ming augmented with type information. This is not a coincidence, as SYNQUID solves constraints using a Horn solver, similar to Prolog. However, where Prolog's execution strategy is straightforward and consistent, SYNQUID's is muddled with subtyping and abstract refinements.

We therefore see that proper usage of SYNQUID requires intimate knowledge with its constraint generation strategy. Users must reframe standard algorithms in terms of refinements that SYNQUID can synthesize, a task that is not trivial. Take the definition of a reverse function:

```
reverse::<r::α → α → Bool>. xs: RList α <r> →
  {RList α <λxλy. r y x> | len ν = len xs}
```

The user first needs to realize that a list is insufficient for synthesizing `reverse`, and that we must augment this data type with a relation `r` that holds between any in-order pair of elements. Then, they must recognize that applying this relation backwards leads to an order reversal. Moreover, we must further understand that since SYNQUID assumes the most restrictive case, then this applies to any relation including $r := \texttt{true}$. Thus, our specification of `reverse` accepts all lists regardless of any relation between their elements. The argument `r` is therefore a required abstraction that obfuscates user intent.

This mental gymnastics required to frame problems in terms of refinement types, once again shows how this programming model is impractical for complex programs. Thus, beyond small scale programs, SYNQUID would only be of use as an interactive aid for completion of code snippets.

SIMBA also leads to a poor user experience, albeit for different reasons. As previously mentioned, to capture the full semantics of a program using SIMBA, we must provide as examples every possible edge case. Clearly, this is very tedious and error prone, especially in the context of complex data types. Take the example of insertion into a balanced binary tree. Framing this problem as a set of input-output pairs requires the inclusion of all possible edge cases, each with a corresponding specification of both input and output trees. This places a burden on the user to be intimately acquainted with the problem they are trying to synthesize, at which point, they are likely capable of efficiently providing an implementation themselves.

Even if we are able to find a proper abstract domain that SIMBA can use to synthesize a complex problem domain, we find ourselves at another bottleneck of providing sufficient input-output pairs. This leaves SIMBA as a tool that is only practical when we can automatically generate examples such as when optimizing an existing procedure.

In conclusion, we have seen how program synthesis as implemented in SYNQUID and SIMBA is greatly restricted in expressiveness to specific problem domains. Moreover, from a user experience standpoint, they do not scale well with complexity. In SYNQUID's case, its generation strategy is highly unintuitive for users, while SIMBA requires an unmanageable volume of examples. We should therefore only expect to use program synthesis as assistants for filling in snippets of simply-structured code, and improving existing code.