Augmenting Type Signatures for Program Synthesis

(Extended Abstract)

Bruce Collie
University of Edinburgh
bruce.collie@ed.ac.uk

Michael O'Boyle
University of Edinburgh
mob@inf.ed.ac.uk

Abstract

Effective program synthesis requires a way to minimise the number of candidate programs being searched. A type signature, for example, places some small restrictions on the structure of potential candidates. We introduce and motivate a distilled program synthesis problem where a type signature is the only machine-readable information available, but does not sufficiently minimise the search space. To address this, we develop a system of property relations that can be used to flexibly encode and query information that was not previously available to the synthesiser. Our experience using these tools has been positive: by encoding simple properties and by using a minimal set of synthesis primitives, we have been able to synthesise complex programs in novel contexts.

1 Motivation

Program synthesis addresses the problem of *automatically* generating correct programs, rather than writing them by hand. A specification constrains the allowable behaviour of a solution; the structure of generated programs and their specifications varies between different synthesis problems.

Our central research aim is to better exploit heterogenous hardware and libraries in user code. Ideally, the compiler should automatically match user code to the best compatible device or library. Our methodology is to automatically learn a formal model of library behaviour that can be used to discover compatible code. We do this by applying program synthesis to functions in libraries with a C interface, which gives us the specification:

> Given a type signature $T \triangleq (\tau_0, \ldots, \tau_n) \to \tau_r$ and a function $f : T$, a correct candidate program $g : T$ is one that for all correctly typed lists of input arguments $\mathbf{x}$, $f(\mathbf{x}) = g(\mathbf{x})$.

In the context of this problem, we are limited to the C type system for compatibility with these existing library interfaces: we have a set of concrete types (`int`, `float`, etc.), along with type constructors for pointers (`int*`), aggregates (`struct{int x; int y;}`) and arrays (`int[10]`).

Unfortunately, the synthesis problem corresponding to this specification is intractable given only the type signature and behaviour of a function. The space of potential solutions is too large to perform any kind of practical search—we need some way of reducing its size. Oracle-guided inductive synthesis [3, 4] makes use of additional information provided by the function oracle to do this (for example, by providing minimal counterexamples to a possible solution), but our black-box C functions do not lend themselves well to this model.

The only formal description we have available for a library function is its type signature, but informal (i.e. human- rather than machine-readable) information can also be used to help the synthesiser. Special-casing individual sources of information in the synthesiser is not scalable—ideally, it would be encoded more formally alongside the type signature to allow the synthesiser to make general use of them during the synthesis process.

2 Property Relations

The solution we propose is to augment function type signatures with a set of property relations over the function's parameters and a set of literal values. This representation is in the vein of a simple logic programming language and allows for flexible encodings of human knowledge.

Our formal definition of these relations is as follows: Let f be a function with type signature $T \triangleq (\tau_0, \ldots, \tau_n) \to \tau_r$, taking parameters $(p_0 : \tau_0, \ldots, p_n : \tau_n)$. Then, define:

$$
\begin{aligned}
P &\triangleq \{p_0, \ldots, p_n\} \\
C &\triangleq \text{set of all C types} \\
S &\triangleq \text{set of all string literals} \\
N &\triangleq \text{set of all numeric literals} \\
U &\triangleq P \cup C \cup S \cup N
\end{aligned}
$$

f is then associated with a set of relations R_f. Each relation $r_i \in R_f$ satisfies $r_i \subseteq U^k$ for some $k > 0$, and has a uniquely identifying name $I(r_i) \in S$ associated with it.

Less formally, named relations group sets of "atoms", where those atoms can be function parameters, literal values or C types. Relations are associated with a function and its type signature, and no particular semantics is attached to them initially (the synthesiser supplies an interpretation for the relations it is given for a function).

Our specification for these relations is intentionally simple—it is the smallest definition we found that would allow for sufficiently useful properties to be encoded. Additionally, a

TyDe'19, August 18, 2019, Berlin, Germany
2019.

close relationship to the function's type signature is maintained: both the signature and the associated relations express facts about a function's behaviour and the meaning attached to its parameters.

3 Synthesis and Queries

The core methodology our program synthesiser is built on is *component-based* synthesis [2, 5], where candidate programs are composed from libraries of smaller fragments. A full description of our synthesis algorithm is outside the scope of this paper, but a brief summary (without considering optimisations or search strategies) is as follows:

- First, a set of potential program fragments is assembled, using the type signature and property annotations to select ones most likely to be present in a correct program.
- Then, an iterative-deepening search enumerates valid compositions of fragments. Some fragments do not compose with others, depending on their context.
- For each composition, instruction sequences are sampled at set program locations specified by the fragment. Each resulting program is JIT-compiled and tested against the reference function.

This method is in the spirit of *two-phase* sketching synthesis [6], where an abstract or partial solution is synthesised first, and is then instantiated to create a full solution.

Given a set of relations associated with a type signature, the synthesiser uses a library of general heuristic patterns to bias its search towards more likely programs. The synthesiser contains a set of fragment "templates", which are partial programs parameterised on values $\in U$ (as defined above), along with rule-based heuristics for their instantiation, written using a simple query language. Fragments provide a control flow structure that can have further fragments nested inside it, or some specific sequence of data-flow instructions likely to occur in a solution.

The queries used to govern fragment instantiation follow the style of a simple logic programming language: a matching expression $r(X, Y)$ is satisfied if the relation named r in the current set is present, and contains a pair of values that can be unified to the variables X, Y. For a single match this is trivial; but conjunctions may be formed, leading to more complex unifications. Additionally, negative matches can be used (but require a conjunction with a positive match to unify). Finally, a standard set of queries can be made of the function's type signature. An example rule for instantiating a joint iteration is:

$$\begin{aligned} &\mathit{size}(X, N) \ \wedge \mathit{size}(Y, N) \ \wedge \\ &\mathit{type}(X, T) \ \wedge \mathit{type}(Y, S) \ \wedge \ \mathit{type}(N, \texttt{int}) \\ &\mathit{is\text{-}pointer}(T) \wedge \mathit{is\text{-}pointer}(S) \\ \implies &\texttt{zip\_loop}(N, T, X, S, Y) \end{aligned}$$

4 Worked Example

An illustrative example of how our approach helps to model and understand performance libraries is the BLAS [1] standard. By synthesizing equivalent programs to functions in BLAS, we have been able to discover opportunities for better library usage in existing scientific code.

The GEMV function in BLAS performs a general matrix-vector multiplication ($\mathbf{y} \leftarrow \alpha\mathbf{A}\mathbf{x} + \beta\mathbf{y}$). It is a challenging target for program synthesisers: it contains nested control flow, 7 input parameters and complex array indexing expressions. For the signature:

```
void gemv(int m, int n, float alpha, float *a,
          float *x, float beta, float *y);
```

we provide the minimal set of property annotations:

$$\{\mathit{size}(x, n), \mathit{size}(y, m), \mathit{output}(y)\}$$

Our synthesiser matches a number of general rules against these properties to reach a correct solution for GEMV. The first is a general loop rule, which matches both `x` and `y`:

$$\begin{aligned} &\mathit{size}(X, N) \wedge \mathit{type}(N, \mathit{int}) \wedge \mathit{type}(X, T) \\ &\wedge \mathit{is\text{-}pointer}(T) \implies \texttt{loop}(N, T, X) \end{aligned}$$

Two loops are instantiated and added to the set of potential fragments. The next rule is one to perform a store to elements of `y`:

$$\mathit{output}(X) \wedge \mathit{type}(X, T) \implies \texttt{store}(X, T)$$

Finally, a negative-match rule matches `a`:

$$\begin{aligned} &\overline{\mathit{size}(X, \_)} \wedge \mathit{type}(X, T) \wedge \mathit{is\text{-}pointer}(T) \\ &\implies \texttt{affine\_access}(X, T) \end{aligned}$$

Other rules are present in the synthesiser, but these are the only ones that match the properties associated with GEMV. The fragments instantiated by the successful matches can compose in such a way that the algorithmic core of GEMV is realised. After this, all that remains is to search for the correct sequence of dataflow instructions:

```
for(int i = 0; i < m; ++i) {
  // code...
  for(int j = 0; j < m; ++j) {
    float v = x[?];
    // code...
  }
  y[i] = ?;
}
```

Our synthesis results are promising: by using minimal property annotations obtained from documentation, together with general-purpose heuristic queries, we have been able to synthesise a wide variety of programs. The synthesised programs come from a number of domains, and have led (with other work) to significant performance improvements on real world scientific programs.

References

[1] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. 2001. An Updated Set of Basic Linear Algebra Subprograms (BLAS). *ACM Trans. Math. Software* 28 (2001), 135–151.

[2] Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-Free Programs. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 62–73. https://doi.org/10.1145/1993498.1993506

[3] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1 (ICSE '10)*. ACM, New York, NY, USA, 215–224. https://doi.org/10.1145/1806799.1806833

[4] Susmit Jha and Sanjit A. Seshia. 2015. A Theory of Formal Synthesis via Inductive Learning. *arXiv:1505.03953 [cs]* (May 2015). arXiv:cs/1505.03953 http://arxiv.org/abs/1505.03953

[5] Yoad Lustig and Moshe Y. Vardi. 2009. Synthesis from Component Libraries. In *Proceedings of the 12th International Conference on Foundations of Software Science and Computational Structures: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009 (FOSSACS '09)*. Springer-Verlag, Berlin, Heidelberg, 395–409. https://doi.org/10.1007/978-3-642-00596-1_28

[6] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Synthesizing Highly Expressive SQL Queries from Input-Output Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017)*. ACM, New York, NY, USA, 452–466. https://doi.org/10.1145/3062341.3062365