NSYNC: Normalised Synthesis of Computation

Zoey Shepherd Ohad Kammar Elizabeth Polgreen

School of Informatics, University of Edinburgh

Typical program synthesis: a needle in a haystack

True, False, a, b,
if True then True else True
if True then True else False
if True then True else a
if True then True else b
if True then False else True
if True then False else False
if True then False else a
if True then False else b
if True then a else True
if True then a else False
if True then a else a
if True then a else b
if True then b else True
if True then b else False
if True then b else a
⋮
if a then True else True
if a then True else False
if a then True else a
if a then True else b
if a then False else True
if a then False else False
if a then False else a
if a then False else b
if a then a else True
if a then a else False
if a then a else a
if a then a else b
if a then b else True
if a then b else False
if a then b else a
⋮

Identifying duplicates →

True, False, a, b
True | True
True | a
True | a
True | if a then True else b
False | if a then False else True
False | False
False | False
False | if a then False else b
a | True
a | a
a | a
a | if a then True else b
b | if a then b else True
b | if a then b else False
b | if a then b else False
⋮ ⋮

Removing duplicates →

True, False, a, b
if a then True else b
if a then False else True
if a then False else b
if a then b else True
if a then b else False
⋮

Program synthesis is the problem of constructing a concrete program that meets some specification. For example, we can specify a program using a set of input-output examples:

$$\{\,(a = \texttt{True}, b = \texttt{True}) \mapsto \texttt{True},\ (a = \texttt{True}, b = \texttt{False}) \mapsto \texttt{False},\ (a = \texttt{False}, b = \texttt{True}) \mapsto \texttt{False},\ (a = \texttt{False}, b = \texttt{False}) \mapsto \texttt{False}\,\} : \mathrm{Ex}[a : \texttt{Bool}, b : \texttt{Bool}; \texttt{Bool}].$$

Typical program synthesis algorithms [2] work by searching a space of possible programs to find a correct one. These approaches search over syntax, producing a search space like the list on the left. But we can clearly see that many of these programs are actually equivalent to ones we've seen before, clogging up our search space with **semantic duplicates**. We can see on the right what our search space would look like if these semantic duplicates were removed, and our target term is much easier to identify.

Our approach: synthesis by semantics

$[\![\texttt{True}]\!]$, $[\![\texttt{False}]\!]$, $[\![a]\!]$, $[\![b]\!]$
$[\![$if a then True else $b]\!]$
$[\![$if a then False else True$]\!]$
$[\![$if a then False else $b]\!]$
$[\![$if a then b else True$]\!]$
$[\![$if a then b else False$]\!]$
$[\![$if b then False else True$]\!]$
$[\![$if b then False else $a]\!]$
$[\![$if b then a else True$]\!]$

By synthesising in a space of semantic objects instead of concrete programs, we can avoid not only the time cost for evaluating semantic duplicates, but also the time cost for constructing them. This gives us a semantically optimal search space, where searching for the correct program can be significantly faster.

Here, we present NSYNC, a proof of concept for our synthesis-by-semantics approach.

Language fragments and STLC+

$$\frac{\Gamma \vdash t : \tau_1 + \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash t_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau}{\Gamma \vdash \delta(t,\ x_1.t_1,\ x_2.t_2) : \tau}$$

Normal forms for a language like Python are undecidable, so we focus on **language fragments** instead. A fragment is a subset of a language which covers a specific semantics, allowing us to reason about that semantics in isolation.

The **Simply-Typed Lambda Calculus with Sums** (STLC+) is one such fragment, having variables, anonymous functions, pairs, a unit type 1 with sole value $\langle\rangle$, and a sum type constructor with injections and match terms. Left and right injections are written $\texttt{in}_L(t)$ and $\texttt{in}_R(t)$, and match terms are written $\delta(t,\ x_1.t_1,\ x_2.t_2)$:

$\delta(t,$	match t with	scrutinee
$x_1.t_1,$	$\texttt{in}_L(x_1) \Rightarrow t_1$	x_1 bound in left branch
$x_2.t_2)$	$\texttt{in}_R(x_2) \Rightarrow t_2$	x_2 bound in right branch.

The typing rule for match terms is shown at the top of the block. We will use this as an example to show how this refines into our normalisation and synthesis rules. The specific refinements will be boxed.

Application: Boolean logic

We can use STLC+ to express Boolean logic: the type Bool is $1 + 1$, and terms $\texttt{True} = \texttt{in}_L\langle\rangle$, $\texttt{False} = \texttt{in}_R\langle\rangle$, and if t then t_1 else $t_2 = \delta(t,\ x_1.t_1,\ x_2.t_2)$. Then the target term from the top desugars to

$$\texttt{if}\ a\ \texttt{then}\ b\ \texttt{else False} = \delta(a,\ x_1.b,\ x_2.\texttt{in}_R\langle\rangle).$$

We don't need to write x_1 and x_2 in the sugared form, since both are of type 1 and therefore have fixed value $\langle\rangle$.

Semantics and normal forms

$$\frac{\Gamma \vdash_{\mathrm{M}} M : \tau_1 + \tau_2 \quad \Gamma, x_1 : \tau_1 \vdash_{\mathrm{N}} N_1 : \tau \quad \Gamma, x_2 : \tau_2 \vdash_{\mathrm{N}} N_2 : \tau \quad \forall i \in \{1,2\}. N_i = \delta(M_i, \dots) \implies \Gamma, x : \tau_i \vdash M \sqsubset M_i \quad N_1 \neq N_2 \text{ unless } x_1 \in FV(N_1) \wedge x_2 \in FV(N_2)}{\Gamma \vdash_{\mathrm{N}} \delta(M,\ x_1.N_1,\ x_2.N_2) : \tau}$$

We want a space of semantic objects, one for each set of semantically equivalent programs. Normalisation by Evaluation assigns to each set a single representative program—a **normal form**—which suffices as this object.

We use normal forms of STLC+ from Balat et al. [3]. They normalise over β-η laws and the Strong Sum Extensionality Axiom

$$\delta(t,\ x_1.t'[\texttt{in}_L x_1/x],\ x_2.t'[\texttt{in}_R x_2/x]) = t'[t/x] \qquad \text{(SSEA)}$$

which removes redundant matches. We extend this by imposing an ordering $\sqsubset$ on scrutinees, giving unique normal forms. One rule for the language of normal forms is shown above.

Normalising Boolean terms

In our Boolean language, the β laws and SSEA are the axioms

$$\texttt{if True then}\ t_1\ \texttt{else}\ t_2 \simeq t_1 \qquad \texttt{if False then}\ t_1\ \texttt{else}\ t_2 \simeq t_2$$
$$\texttt{if}\ c\ \texttt{then}\ t\ \texttt{else}\ t \simeq t$$

which are enough to derive all of the equalities in the worked example at the top. Some of the normal forms are

$$[\![\texttt{True}]\!] = \texttt{True} \qquad [\![\texttt{False}]\!] = \texttt{False}$$
$$[\![a]\!] = \texttt{if}\ a\ \texttt{then True else False}$$
$$[\![\texttt{if}\ b\ \texttt{then}\ a\ \texttt{else False}]\!] = \texttt{if}\ a\ \texttt{then}\ (\texttt{if}\ b\ \texttt{then True else False})\ \texttt{else False}.$$

The final equality uses our ordering $a : \texttt{Bool}, b : \texttt{Bool} \vdash a \sqsubset b$ to choose the order of branches, independently of the original term.

Normalised synthesis

$$\frac{[\Gamma; \tau_1 + \tau_2], c \rightsquigarrow_{\mathrm{M}} M \quad X_1 \triangleq \{\sigma \cdot [v_1/x_1] \mapsto X(\sigma) \mid \sigma \in \mathrm{dom}(X), M[\sigma] \to^* \texttt{in}_L(v_1)\} \quad X_2 \triangleq \{\sigma \cdot [v_2/x_2] \mapsto X(\sigma) \mid \sigma \in \mathrm{dom}(X), M[\sigma] \to^* \texttt{in}_R(v_2)\} \quad X_1 \neq \emptyset \quad X_2 \neq \emptyset \quad X_1 : \mathrm{Ex}[\Gamma, x_1 : \tau_1; \tau], (M \sqsubset) \rightsquigarrow_{\mathrm{N}} N_1 \quad \text{if } x_1 \notin FV(N_1), \text{ then } \exists \sigma \in \mathrm{dom}(X_2) \text{ s.t. } N_1[\sigma] \not\to^* X_2(\sigma) \quad X_2 : \mathrm{Ex}[\Gamma, x_2 : \tau_2; \tau], (M \sqsubset) \rightsquigarrow_{\mathrm{N}} N_2}{X : \mathrm{Ex}[\Gamma; \tau], c \rightsquigarrow_{\mathrm{N}} \delta(M,\ x_1.N_1,\ x_2.N_2)}$$

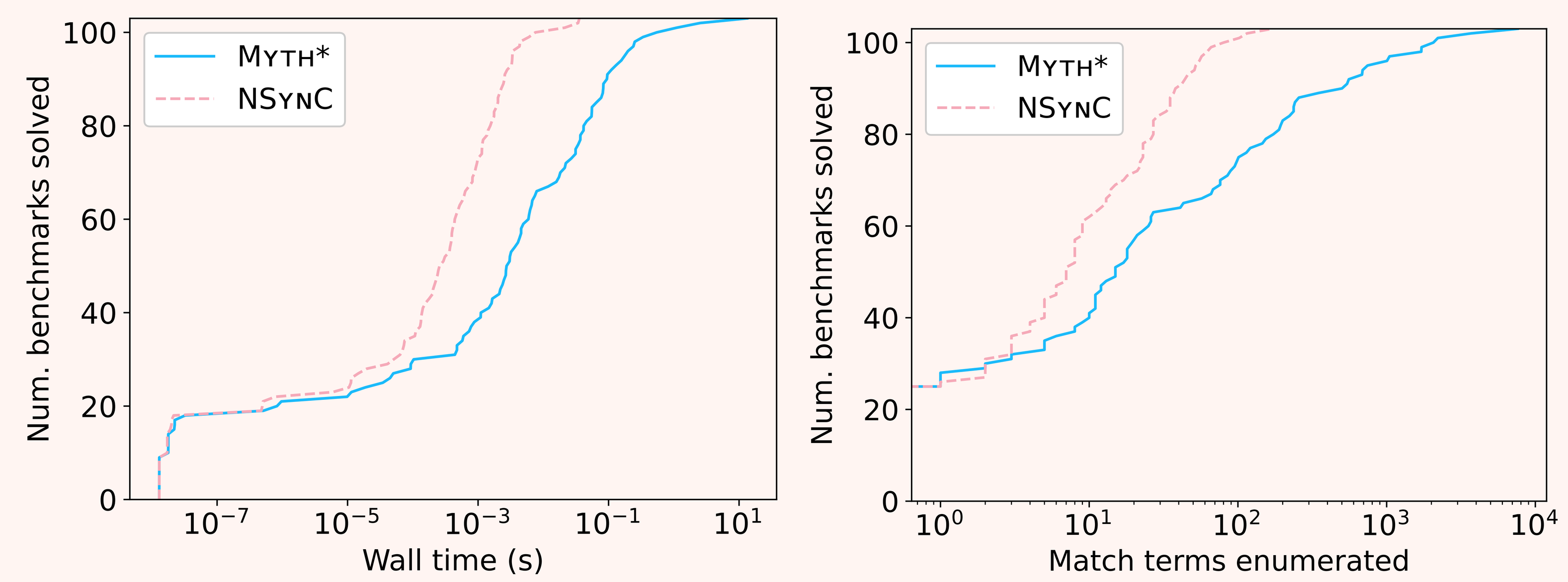
Our synthesis algorithm is based on MYTH [4]. Both methods iteratively apply a set of synthesis rules, and the difference between these rules is the core of our approach. Our rule for match terms, shown above, is derived from the normalisation rule, whereas MYTH's comes directly from the definition of STLC+.

The synthesis rule involves manipulating an example set X, which is typed with a context and a goal type. For match terms, we separate the examples into two subsets, one for each branch, and bind the new variable in each one.

We rearrange the extra conditions of normal forms to be more amenable to synthesis. The ordering condition becomes a constraint that can be passed recursively. To avoid synthesising N_2 unnecessarily, we make the check for identical branches preemptive. The separation of terms into sorts is already included in MYTH, so we simply inherit it.

The NSYNC algorithm exhaustively applies the synthesis rules until a solution emerges. To avoid non-termination in any subproblem, we impose size limits on neutral terms—that is, scrutinees and terms of base type—and a limit on the depth of match terms. The user inputs a search procedure over these limits, gradually increasing them until NSYNC finds a satisfying term.

Preliminary results



We evaluate NSYNC on 166 synthetic benchmarks and see a geomean speedup of **8.93x** over our re-implementation of MYTH. Above are cactus plots giving the time taken and number of match terms enumerated by each algorithm for each of the 104 benchmarks that both solved.

We also show the following properties of NSYNC:

Bounded Completeness. If the normal form of a term t does not have any branching argument—a function that branches on its input and is given as an argument to a higher-order function—then NSYNC can produce that normal form. Since MYTH has the same restriction, restricting to normal forms maintains solvability.

Semantic Optimality. All of NSYNC's candidate terms are semantically distinct.

Where do we go from here?

We aim to extend NSYNC from a proof of concept to a practical synthesis method. We will consider language features like lists [1] or universal algebras [5] using corresponding fragments. We then want to explore general-purpose synthesisers as combinations of these specialised synthesis-by-semantics components.

References

[1] Guillaume Allais, Conor McBride, and Pierre Boutillier. New equations for neutral terms: a sound and complete decision procedure, formalized. In *Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming*, pages 13–24, 2013.

[2] Rajeev Alur, Rastislav Bodik, Garvit Juniwal, Milo MK Martin, Mukund Raghothaman, Sanjit A Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-guided synthesis. In *2013 Formal Methods in Computer-Aided Design*, pages 1–8. IEEE, 2013.

[3] Vincent Balat, Roberto Di Cosmo, and Marcelo Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. *ACM SIGPLAN Notices*, 39(1):64–76, 2004.

[4] Peter-Michael Osera and Steve Zdancewic. Type-and-example-directed program synthesis. *ACM SIGPLAN Notices*, 50(6):619–630, 2015.

[5] Jeremy Yallop, Tamara Von Glehn, and Ohad Kammar. Partially-static data as free extension of algebras. *Proceedings of the ACM on Programming Languages*, 2(ICFP):1–30, 2018.

Submitted to SYNT 2026

z.shepherd@ed.ac.uk