NSynC: Normalised Synthesis of Computation

Zoey Shepherd, Ohad Kammar, and Elizabeth Polgreen

University of Edinburgh, Edinburgh, UK

Abstract. Inductive program synthesis algorithms search a space of programs to find one that meets some specification. Enumerating according to the syntax of a programming language leads to a large search space, and hence slow synthesis, due in large part to *semantic duplication*. A synthesiser may have to evaluate—and reject—multiple semantically identical but syntactically different programs, wasting resources.
To avoid this duplication, we present NSynC, a synthesis-by-semantics approach. By enumerating the semantics of the target language directly, we guarantee that each candidate program is semantically unique and that each evaluation of a candidate is meaningful. Specifically, we search the space of normal forms for the simply-typed lambda calculus with sums using a top-down, type-directed synthesis algorithm. Our preliminary results show a geomean speedup of 8.93x on a synthetic benchmark suite over the unrestricted algorithm.

Introduction. Program synthesis is the task of taking a program specification to a concrete program that meets it. Deductive approaches (e.g. [6]) are limited in what programs they can produce, so search-based approaches like Counter-Example Guided Inductive Synthesis [8] are more common for modern program synthesis [3]. These search-based approaches search over *syntax*. That is, if the syntax of the target language includes the production rule for Booleans $\texttt{Bool} ::= \texttt{True} \mid \texttt{False} \mid a \mid b \mid \texttt{Bool} \,\&\, \texttt{Bool} \mid \neg\texttt{Bool}$, then a naïve synthesiser will consider the syntactically distinct terms $a \,\&\, b$, $b \,\&\, a$, $a \,\&\, (b \,\&\, \texttt{True})$, $\neg(\neg(a \,\&\, b))$, and so on to be distinct "guesses" for a target program of type `Bool`. Of course, all of these programs are semantically identical, but a synthesiser might have to consider each of them individually to be able to discount them all.

Many synthesisers take steps to reduce the impact of semantic duplication. The standard formulation of bottom-up enumeration [9] removes observationally equivalent programs from its pool at each step, preventing duplicates from surviving for more than one iteration and avoiding construction of terms like $a \,\&\, (b \,\&\, \texttt{True})$—but not, say, $a \,\&\, b$ and $b \,\&\, a$. In Semantics-Guided Synthesis [5], semantics are provided to the solver as a logical specification, making the process of ruling out duplicates easier but leaving them in the search space.

Our Approach. To fully eliminate semantic duplication, an algorithm would search the space of *semantics*, rather than syntax. We propose leveraging Normalisation by Evaluation, which constructs unique normal forms for semantic fragments of languages, to provide this search space. As a proof of concept, we present NSynC (Normalised Synthesis of Computation), which synthesises

terms of the simply-typed lambda calculus with sums (STLC+) restricted to unique normal forms based on Balat et al. [4]. NSynC uses the same top-down, type-directed framework as Myth [7].

STLC+ is a simple but powerful language fragment with a lot of semantic redundancy. It extends STLC, which has anonymous functions, with pairs, a unit type 1 with sole value $\langle\rangle$, and sum types $\tau_1 + \tau_2$ equipped with left and right injections $\mathtt{in}_L^{\tau_1+\tau_2} t$ and $\mathtt{in}_R^{\tau_1+\tau_2} t$ and match terms $\delta(t,\, x_1.t_1,\, x_2.t_2)$ analogous to

$\mathtt{match}\; t\; \mathtt{with}$		t is the scrutinee
$\quad \mathtt{in}_L x_1 \Rightarrow t_1$	where	t_1 is the left branch, binding x_1
$\quad \mathtt{in}_R x_2 \Rightarrow t_2$		t_2 is the right branch, binding x_2.

We can express Boolean logic using STLC+: type $\mathtt{Bool} \triangleq 1 + 1$, and terms $\mathtt{True} \triangleq \mathtt{in}_L^{\mathtt{Bool}}\langle\rangle$, $\mathtt{False} \triangleq \mathtt{in}_R^{\mathtt{Bool}}\langle\rangle$, $\mathtt{if}\, t\, \mathtt{then}\, t_1\, \mathtt{else}\, t_2 \triangleq \delta(t,\, x_1.t_1,\, x_2.t_2)$, and $a \,\&\, b \triangleq \mathtt{if}\, a\, \mathtt{then}\, b\, \mathtt{else}\, \mathtt{False}$.

Myth already eliminates some of the semantic duplication in STLC+ by synthesising only β-normal, η-long forms—where every computation is fully simplified and every term is expanded into its maximal structural form. This allows Myth to skip terms like $\mathtt{if}\, \mathtt{True}\, \mathtt{then} \ldots$, but it doesn't fully cover the semantics of match terms, which induce several semantic duplicates like $\mathtt{if}\, a\, \mathtt{then}\, t\, \mathtt{else}\, t = t$ or $a \,\&\, b = b \,\&\, a$.

Balat et al. [4] account for both β-η equivalences—as Myth does—and the strong sum extensionality axiom, which is responsible for equalities like above. Their normal forms eliminate redundant branches like $\mathtt{if}\, a\, \mathtt{then}\, t\, \mathtt{else}\, t$, and our extension with an ordering on scrutinees chooses between, e.g., $a \,\&\, b$ and $b \,\&\, a$, giving unique normal forms (proof in Appendix B). These normal forms define a fragment of STLC+ with full expressibility and no semantic duplication, giving an ideal search space for synthesis.

NSynC, like Myth, iteratively applies a set of synthesis rules. The derivation of these rules is the core of our approach: a direct application of Myth to STLC+ gets its rules directly from the language's definition, whereas NSynC draws them from the rules on normal forms. One such rule, for synthesising match terms, is

$$\frac{\begin{array}{c}[\Gamma; \tau_1 + \tau_2], \boxed{c} \rightsquigarrow_{\mathrm{M}} M \\ X_1 \triangleq \{\sigma \cdot [v'/x_1] \mapsto v \mid \sigma \mapsto v \in X, M[\sigma] \to^* \mathtt{in}_L^{\tau_1+\tau_2}(v')\} \\ X_2 \triangleq \{\sigma \cdot [v'/x_2] \mapsto v \mid \sigma \mapsto v \in X, M[\sigma] \to^* \mathtt{in}_R^{\tau_1+\tau_2}(v')\} \\ \forall i \in \{1,2\}.\; X_i \neq \emptyset \wedge X_i : \mathrm{Ex}[\Gamma, x_i : \tau_i; \tau], \boxed{(M \sqsubset)} \rightsquigarrow_{\mathrm{N}} N_i \\ \boxed{x_1 \notin FV(N_1) \wedge x_2 \notin FV(N_2) \implies N_1 \neq N_2}\end{array}}{X : \mathrm{Ex}[\Gamma; \tau], \boxed{c} \rightsquigarrow_{\mathrm{N}} \delta(M,\, x_1.N_1,\, x_2.N_2)}$$

which begins with producing a scrutinee M of sum type, then separates the example set X accordingly to synthesise branches N_1 and N_2. The boxes highlight our divergence from Myth: the constraints c and $M \sqsubset$ force scrutinees to appear in our chosen order, and the final precondition forces the two branches to be distinct. We show how our other rules differ from Myth in Appendix C.2.

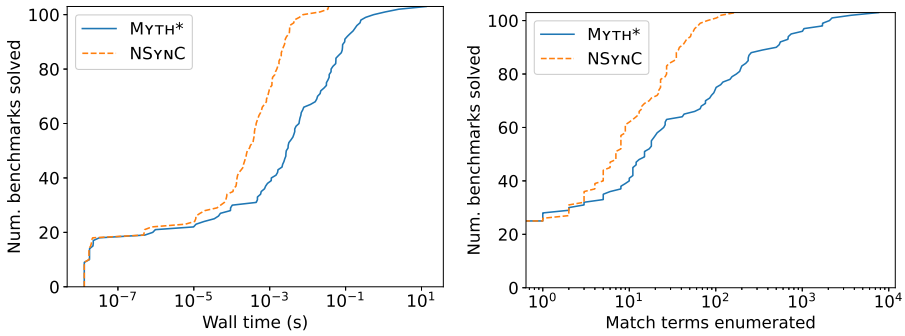

Fig. 1: Log-scaled cactus plot showing the number of benchmarks that could be solved within time (left) and enumeration (right) constraints.

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.

Theoretical Results. We show *bounded correctness* of NSynC in Appendix C.3. Specifically, we show *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, we have that restricting to normal forms maintains solvability. We also show *semantic optimality*: all of NSynC's candidate terms are semantically distinct.

Empirical Evaluation. We evaluate NSynC on a set of 166 randomly-generated synthetic benchmarks and see a geomean speedup of 8.93x. We compare NSynC with Myth*, our re-implementation of Myth. Figure 1 shows, for each of the 104 benchmarks that both algorithms solved, the time taken and the number of match terms enumerated by each algorithm. A further 37 benchmarks were solved by Myth* but not NSynC, since enforcing an order on scrutinees sometimes increases the number of branches in a program so that the example set is unable to cover them; in future work, we aim to address this problem using a parallel case split construction from Altenkirch et al. [2].

Future Work. After addressing this limitation, the next step is to bring NSynC from a proof of concept to a practical synthesis method. In particular, we aim to extend our analysis to different language fragments that have normal forms, such as STLC with lists [1], or fragments corresponding to universal algebras [10]. We then want to explore general-purpose synthesisers as combinations of these specialised, synthesis-by-semantics components.

Acknowledgments. Funding for this research was provided by EPSRC through a PhD studentship within the CDT in Machine Learning Systems hosted in the School of Informatics, University of Edinburgh (EP/Y03516X/1).

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Allais, G., McBride, C., Boutillier, P.: New equations for neutral terms: a sound and complete decision procedure, formalized. In: Proceedings of the 2013 ACM SIGPLAN Workshop on Dependently-typed Programming. pp. 13–24 (2013)
2. Altenkirch, T., Dybjer, P., Hofmann, M., Scott, P.: Normalization by evaluation for typed lambda calculus with coproducts. In: Proceedings 16th Annual IEEE Symposium on Logic in Computer Science. pp. 303–310. IEEE (2001)
3. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design. pp. 1–8. IEEE (2013)
4. Balat, V., Di Cosmo, R., Fiore, M.: Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. ACM SIGPLAN Notices **39**(1), 64–76 (2004)
5. Kim, J., Hu, Q., D'Antoni, L., Reps, T.: Semantics-guided synthesis. Proceedings of the ACM on Programming Languages **5**(POPL), 1–32 (2021)
6. Manna, Z., Waldinger, R.: Fundamentals of deductive program synthesis. IEEE Transactions on Software Engineering **18**(08), 674–704 (1992)
7. Osera, P.M., Zdancewic, S.: Type-and-example-directed program synthesis. ACM SIGPLAN Notices **50**(6), 619–630 (2015)
8. Solar-Lezama, A., Tancau, L., Bodik, R., Seshia, S., Saraswat, V.: Combinatorial sketching for finite programs. In: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems. pp. 404–415 (2006)
9. Udupa, A., Raghavan, A., Deshmukh, J.V., Mador-Haim, S., Martin, M.M., Alur, R.: Transit: specifying protocols with concolic snippets. ACM SIGPLAN Notices **48**(6), 287–296 (2013)
10. Yallop, J., Von Glehn, T., Kammar, O.: Partially-static data as free extension of algebras. Proceedings of the ACM on Programming Languages **2**(ICFP), 1–30 (2018)

$$
\begin{array}{rcll}
\tau & ::= & \theta_i & \text{base types} \\
& | & 1 & \text{unit type} \\
& | & \tau \to \tau & \text{function types} \\
& | & \tau \times \tau & \text{product types} \\
& | & \tau + \tau & \text{sum types}
\end{array}
$$

$$
\begin{array}{rl}
\texttt{Bool} & = 1 + 1 \\
\texttt{True} & = \texttt{in}_L^{\texttt{Bool}}\langle\rangle \\
\texttt{False} & = \texttt{in}_R^{\texttt{Bool}}\langle\rangle \\
\texttt{if}\, t\, \texttt{then}\, t_1\, \texttt{else}\, t_2 & = \delta(t,\, x_1.t_1,\, x_2.t_2)
\end{array}
$$

$$
\begin{array}{rcll}
t & ::= & x & \text{variables} \\
& | & \langle\rangle & \text{unit} \\
& | & \lambda(x:\tau).t & \text{abstraction} \\
& | & t\, t & \text{application} \\
& | & \langle t, t\rangle & \text{pairing} \\
& | & \pi_1 t & \text{first projection} \\
& | & \pi_2 t & \text{second projection} \\
& | & \texttt{in}_L^{\tau+\tau} t & \text{left injection} \\
& | & \texttt{in}_R^{\tau+\tau} t & \text{right injection} \\
& | & \delta(t,\, x.t,\, x.t) & \text{match term}
\end{array}
$$

Fig. 2: Grammar for types (upper left) and terms (right) of STLC+, where i ranges over some index set I of base types and x ranges over variables. We also give syntactic sugar (lower left) for Boolean values.

A STLC+

The simply-typed lambda calculus with sums (STLC+) is the language over the context-free grammar shown in Figure 2. θ_i denotes a base type, indexed by i. 1 is the unit type, containing only the value $\langle\rangle$. $\tau_1 \to \tau_2$ are function types, with $\lambda(x : \tau_1).t$ as the constructor and $t_1\, t_2$ as the application of t_1 to t_2. $\tau_1 \times \tau_2$ are product types, constructed as $\langle t_1, t_2\rangle$ and destructed with π_1 and π_2, which take the first and second element of the pair respectively. $\tau_1 + \tau_2$ are sum types, described in the main paper.

Most formulations of STLC+ (including Balat et al.'s [4]) also have an empty type 0 which contains no values, with the semantics that finding a term of the empty type is a contradiction from which any other type can be derived. By building our types from the grammar in Figure 2, we can avoid any risk of this contradiction arising.

This appendix gives the typing rules (Figure 3), operational semantics (Figure 4), and equational theory (Subappendix A.1) for STLC+.

We write $\Gamma \vdash t : \tau$ for the typing judgement, saying that in typing context Γ the term t is well-typed at τ. A typing context $\Gamma \vdash \cdot \mid \Gamma, x : \tau$ is simply a list assigning types to free variables, and we assume that the variables in a given context are unique.

We give operational semantics in terms of a relation $t \to^* v$, meaning a term t evaluates to value v. Values are simply fully reduced closed terms, given by the grammar $v ::= a \mid \langle\rangle \mid \lambda(x : \tau).t \mid \langle v, v\rangle \mid \texttt{in}_L^{\tau+\tau} v \mid \texttt{in}_R^{\tau+\tau} v$, where a stands for any constant and the remaining symbols are the constructors in the term language.

Note that there is no rule for evaluation under a λ-abstraction. This is because we define equality between function-typed values using extensional equality: when $\cdot \vdash v_1, v_2 : \tau' \to \tau$, we have $v_1 = v_2$ if and only if, for all values $\cdot \vdash v : \tau'$, we have $v_1\, v \to^* w_1$ and $v_2\, v \to^* w_2$ s.t. $w_1 = w_2$.

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{TypeVar} \qquad \frac{}{\Gamma \vdash \langle\rangle : 1}\ \textsc{TypeUnit}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau}{\Gamma \vdash \lambda(x : \tau_1).t : \tau_1 \to \tau}\ \textsc{TypeAbs} \qquad \frac{\Gamma \vdash t_1 : \tau_1 \to \tau \qquad \Gamma \vdash t_2 : \tau_1}{\Gamma \vdash t_1\, t_2 : \tau}\ \textsc{TypeApp}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \langle t_1, t_2\rangle : \tau_1 \times \tau_2}\ \textsc{TypePair}$$

$$\frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_1 t : \tau_1}\ \textsc{TypeFst} \qquad \frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \pi_2 t : \tau_2}\ \textsc{TypeSnd}$$

$$\frac{\Gamma \vdash t : \tau_1}{\Gamma \vdash \mathtt{in}_L^{\tau_1+\tau_2}(t) : \tau_1 + \tau_2}\ \textsc{TypeInL} \qquad \frac{\Gamma \vdash t : \tau_2}{\Gamma \vdash \mathtt{in}_R^{\tau_1+\tau_2}(t) : \tau_1 + \tau_2}\ \textsc{TypeInR}$$

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

Fig. 3: Typing rules for STLC+.

$$\frac{\cdot \vdash v : \tau}{v \to^* v}\ \textsc{EvalValue} \qquad \frac{t_1 \to^* \lambda(x : \tau).t' \qquad t'[t_2/x] \to^* v}{t_1\, t_2 \to^* v}\ \textsc{EvalApp}$$

$$\frac{t_1 \to^* v_1 \;\; t_1 \to^* v_2}{\langle t_1, t_2\rangle \to^* \langle v_1, v_2\rangle}\ \textsc{EvalPair}$$

$$\frac{t \to^* \langle v_1, v_2\rangle}{\pi_1 t \to^* v_1}\ \textsc{EvalFst} \qquad \frac{t \to^* \langle v_1, v_2\rangle}{\pi_2 t \to^* v_2}\ \textsc{EvalSnd}$$

$$\frac{t \to^* v}{\mathtt{in}_L^{\tau_1+\tau_2} t \to^* \mathtt{in}_L^{\tau_1+\tau_2} v}\ \textsc{EvalInL} \qquad \frac{t \to^* v}{\mathtt{in}_R^{\tau_1+\tau_2} t \to^* \mathtt{in}_R^{\tau_1+\tau_2} v}\ \textsc{EvalInR}$$

$$\frac{t \to^* \mathtt{in}_L^{\tau_1+\tau_2}(v') \qquad t_1[v'/x_1] \to^* v}{\delta(t,\, x_1.t_1,\, x_2.t_2) \to^* v}\ \textsc{EvalMatchL}$$

$$\frac{t \to^* \mathtt{in}_R^{\tau_1+\tau_2}(v') \qquad t_2[v'/x_2] \to^* v}{\delta(t,\, x_1.t_1,\, x_2.t_2) \to^* v}\ \textsc{EvalMatchR}$$

Fig. 4: Operational semantics of STLC+.

$$\frac{\Gamma \vdash t : 1}{\Gamma \vdash t \simeq_1 \langle\rangle}\ \textsc{EqUnit}$$

$$\frac{\Gamma, x : \tau_1 \vdash t : \tau \qquad \Gamma \vdash t_1 : \tau_1}{\Gamma \vdash (\lambda(x : \tau_1).t)\, t_1 \simeq_\tau t[t_1/x]}\ \textsc{EqFunc}\beta$$

$$\frac{\Gamma \vdash t : \tau_1 \to \tau}{\Gamma \vdash \lambda(x : \tau_1).(t\, x) \simeq_{\tau_1 \to \tau} t}\ \textsc{EqFunc}\eta$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \pi_1\langle t_1, t_2\rangle \simeq_{\tau_1} t_1}\ \textsc{EqProd}\beta_1 \qquad \frac{\Gamma \vdash t_1 : \tau_1 \qquad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash \pi_2\langle t_1, t_2\rangle \simeq_{\tau_2} t_2}\ \textsc{EqProd}\beta_2$$

$$\frac{\Gamma \vdash t : \tau_1 \times \tau_2}{\Gamma \vdash \langle \pi_1 t, \pi_2 t\rangle \simeq_{\tau_1 \times \tau_2} t}\ \textsc{EqProd}\eta$$

$$\frac{\Gamma \vdash t : \tau_1 \qquad \Gamma, x_1 : \tau_1 \vdash t_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau}{\Gamma \vdash \delta(\mathtt{in}_L^{\tau_1+\tau_2} t,\, x_1.t_1,\, x_2.t_2) \simeq_\tau t_1[t/x_1]}\ \textsc{EqSum}\beta_L$$

$$\frac{\Gamma \vdash t : \tau_2 \qquad \Gamma, x_1 : \tau_1 \vdash t_1 : \tau \qquad \Gamma, x_2 : \tau_2 \vdash t_2 : \tau}{\Gamma \vdash \delta(\mathtt{in}_R^{\tau_1+\tau_2} t,\, x_1.t_1,\, x_2.t_2) \simeq_\tau t_2[t/x_1]}\ \textsc{EqSum}\beta_R$$

$$\frac{\Gamma \vdash t : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 + \tau_2 \vdash t' : \tau}{\Gamma \vdash \delta(t,\, x_1.t'[\mathtt{in}_L^{\tau_1+\tau_2} x_1/x],\, x_2.t'[\mathtt{in}_R^{\tau_1+\tau_2} x_2/x]) \simeq_\tau t'[t/x]}\ \textsc{EqSSEA}$$

Fig. 5: An equational theory for STLC+.

A.1 Semantic Equality

We write $\Gamma \vdash t_1 \simeq_\tau t_2$ to mean that t_1 and t_2 are semantically equal at type τ in context Γ; we may choose to omit the type for brevity. The operational semantics above satisfies this equational theory: if $\Gamma \vdash t_1 \simeq_\tau t_1$, then for all environments $\Gamma \vdash \sigma$ assigning to each variable in Γ a value of the appropriate type, $t_1[\sigma] \to^* v_1$ and $t_2[\sigma] \to^* v_2$ s.t. $v_1 = v_2$.

Semantic equality is inductively defined as the reflexive, symmetric, transitive, and congruence closure of the rules shown in Figure 5.

The first rule EqUnit is simply that the unit type only contains one value. The rest are familiar β-η laws for each type, except for EqSSEA, the Strong Sum Extensionality Axiom [4]. EqSSEA generalises the η law for sum types (recovered by setting $t' = x$) with the added property that execution contexts distribute over match terms.

Example 1. We define the following syntactic sugar:

$$\begin{aligned}
\texttt{Bool} &= 1 + 1 \\
\texttt{True} &= \texttt{in}_L^{\texttt{Bool}}\langle\rangle \\
\texttt{False} &= \texttt{in}_R^{\texttt{Bool}}\langle\rangle \\
\texttt{if}\, t\, \texttt{then}\, t_1\, \texttt{else}\, t_2 &= \delta(t,\, x_1.t_1,\, x_2.t_2) \\
t_1 \& t_2 &= \texttt{if}\, t_1\, \texttt{then}\, t_2\, \texttt{else}\, \texttt{False}
\end{aligned}$$

and show that the axioms above derive $a : \texttt{Bool}, b : \texttt{Bool} \vdash a\&b \simeq_{\texttt{Bool}} b\&a$:

$$\begin{aligned}
a\&b &= \delta(a,\, x_1.b,\, x_2.\texttt{in}_R^{\texttt{Bool}}\langle\rangle) \\
&= \delta(a,\, x_1.\delta(b,\, y_1.\texttt{in}_L^{\texttt{Bool}}y_1,\, y_2.\texttt{in}_R^{\texttt{Bool}}y_2),\, x_2.\delta(b,\, y_1.\texttt{in}_R^{\texttt{Bool}}\langle\rangle,\, y_2.\texttt{in}_R^{\texttt{Bool}}\langle\rangle)) \\
&= \delta(a,\, x_1.\delta(b,\, y_1.\texttt{in}_L^{\texttt{Bool}}x_1,\, y_2.\texttt{in}_R^{\texttt{Bool}}\langle\rangle),\, x_2.\delta(b,\, y_1.\texttt{in}_R^{\texttt{Bool}}x_2,\, y_2.\texttt{in}_R^{\texttt{Bool}}\langle\rangle))
\end{aligned}$$

The first equality is just desugaring, and the second is by two applications of EqSSEA; one to η-expand the left branch, the other to introduce on the right branch a match with identical branches. We then apply EqUnit to re-label the unit-valued terms, and that allows us to set $t' = \delta(b,\, y_1.x,\, y_2.\texttt{in}_R^{\texttt{Bool}}\langle\rangle)$ and apply EqSSEA once more to arrive at our result:

$$\begin{aligned}
\delta(a,\, x_1.t'[\texttt{in}_L^{\texttt{Bool}}x_1/x],\, x_2.t'[\texttt{in}_R^{\texttt{Bool}}x_2/x]) &= t'[a/x] \\
&= \delta(b,\, y_1.a,\, y_2.\texttt{in}_R^{\texttt{Bool}}\langle\rangle) \\
&= b\&a
\end{aligned}$$

B Normal Forms of STLC+

We begin by presenting the normal forms of STLC+ given by Balat et al. [4]. As noted in their paper, these normal forms are not unique. For example,

$$\texttt{if}\, a\, \texttt{then}\, (\texttt{if}\, b\, \texttt{then}\, \texttt{True}\, \texttt{else}\, \texttt{False})\, \texttt{else}\, \texttt{False}, \qquad \text{and}$$
$$\texttt{if}\, b\, \texttt{then}\, (\texttt{if}\, a\, \texttt{then}\, \texttt{True}\, \texttt{else}\, \texttt{False})\, \texttt{else}\, \texttt{False}.$$

are both normal forms of $a\&b$. They suggest that fixing an ordering on scrutinees would give unique normal forms; we present our ordering in Subappendix B.1 and show in Subappendix B.2 that it does, in fact, give uniqueness.

The grammar of normal forms is given in Figure 6, where the semantics of the terminal symbols are unchanged; note that this, and that $P ::= M$ only at base type, already gives β-long, η-short forms. M are neutral terms, those where computation is stuck at a free variable, P are pure normal terms, and N are normal terms. What we call neutral terms are referred to by Balat et al. as "pure neutral", distinguishing from "impure" neutral terms which can include match statements. However, with N as the start symbol, there is no way to reach an "impure" neutral in the grammar, so we do not need to consider them.

We write $\Gamma \vdash_{\mathrm{M}} t : \tau$ to denote that t is a neutral term at type τ in context Γ, and similarly for $\Gamma \vdash_{\mathrm{P}} \ldots$ and $\Gamma \vdash_{\mathrm{N}} \ldots$. The complete rules for normal forms [4, Figure 3] are given in Figure 7; most are identical to the corresponding typing rules, with the exception of NfNeutral—which is restricted to base type—and NfAbs* and NfMatch*.

$$
\begin{aligned}
\tau &::= \theta_i \mid 1 \mid \tau \to \tau \mid \tau \times \tau \mid \tau + \tau \\
M &::= x \mid M\ P \mid \pi_1 M \mid \pi_2 M \\
P &::= M \mid \langle\rangle \mid \lambda(x:\tau).N \mid \langle P, P\rangle \mid \mathtt{in}_L^{\tau+\tau} P \mid \mathtt{in}_R^{\tau+\tau} P \\
N &::= P \mid \delta(M,\ x.N,\ x.N)
\end{aligned}
$$

Fig. 6: Grammar for normal forms of STLC+ [4].

$$\frac{x:\tau \in \Gamma}{\Gamma \vdash_{\mathrm{M}} x:\tau}\ \textsc{NFVar} \qquad \frac{\Gamma \vdash_{\mathrm{M}} M:\tau_1 \to \tau \qquad \Gamma \vdash_{\mathrm{P}} P:\tau_1}{\Gamma \vdash_{\mathrm{M}} M\,P:\tau}\ \textsc{NFApp}$$

$$\frac{\Gamma \vdash_{\mathrm{M}} M:\tau_1 \times \tau_2}{\Gamma \vdash_{\mathrm{M}} \pi_1 M:\tau_1}\ \textsc{NFFst} \qquad \frac{\Gamma \vdash_{\mathrm{M}} M:\tau_1 \times \tau_2}{\Gamma \vdash_{\mathrm{M}} \pi_2 M:\tau_2}\ \textsc{NFSnd}$$

$$\frac{\exists i.\Gamma \vdash_{\mathrm{M}} M:\theta_i}{\Gamma \vdash_{\mathrm{P}} M:\theta_i}\ \textsc{NFNeutral} \qquad \frac{}{\Gamma \vdash_{\mathrm{P}} \langle\rangle:1}\ \textsc{NFUnit}$$

$$\frac{\Gamma, x:\tau_1 \vdash_{\mathrm{N}} N:\tau \qquad \forall C \in \mathrm{Guards}(N).x \in FV(C)}{\Gamma \vdash_{\mathrm{P}} \lambda(x:\tau_1).N:\tau_1 \to \tau}\ \textsc{NFAbs}^*$$

$$\frac{\Gamma \vdash_{\mathrm{P}} P_1:\tau_1 \qquad \Gamma \vdash_{\mathrm{P}} P_2:\tau_2}{\Gamma \vdash_{\mathrm{P}} \langle P_1, P_2\rangle:\tau_1 \times \tau_2}\ \textsc{NFPair}$$

$$\frac{\Gamma \vdash_{\mathrm{P}} P:\tau_1}{\Gamma \vdash_{\mathrm{P}} \mathtt{in}_L^{\tau_1+\tau_2}(P):\tau_1+\tau_2}\ \textsc{NFInL} \qquad \frac{\Gamma \vdash_{\mathrm{P}} P:\tau_2}{\Gamma \vdash_{\mathrm{P}} \mathtt{in}_R^{\tau_1+\tau_2}(P):\tau_1+\tau_2}\ \textsc{NFInR}$$

$$\frac{\Gamma \vdash_{\mathrm{P}} P:\tau}{\Gamma \vdash_{\mathrm{N}} P:\tau}\ \textsc{NFPure}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{M}} M:\tau_1+\tau_2 \\ \forall i \in \{1,2\}.\ \Gamma, x_i:\tau_i \vdash_{\mathrm{N}} N_i:\tau \wedge \\ \forall C \in \mathrm{Guards}(x_i.N_i).M \neq C \\ x_1 \notin FV(N_1) \wedge x_2 \notin FV(N_2) \implies N_1 \neq N_2\end{array}}{\Gamma \vdash_{\mathrm{N}} \delta(M,\ x_1.N_1,\ x_2.N_2):\tau}\ \textsc{NFMatch}^*$$

$$\mathrm{Guards}(N) \triangleq \begin{cases}\{M\} \cup \bigcup_{i\in\{1,2\}} \mathrm{Guards}(x_i.N_i) & \text{if } N = \delta(M,\ x_1.N_1,\ x_2.N_2) \\ \emptyset & \text{otherwise}\end{cases}$$

$$\mathrm{Guards}(x_i.N_i) \triangleq \{C \in \mathrm{Guards}(N) \mid x_i \notin FV(C)\}.$$

Fig. 7: Normal form rules for STLC+ [4]. $FV(t)$ denotes the free variables that appear in a term t.

B.1 Ordering of Neutrals

To get unique normal forms, we replace the guard restriction in NFMatch* with a stronger restriction in terms of an ordering on neutral terms, which we define in this appendix. This also allows us to rephrase the guard restriction in NFAbs* in a similar way.

We write $\Gamma \vdash M_1 \sqsubset M_2$ to mean that M_1 is strictly before M_2 in context Γ. Fixing Γ gives us a binary relation $\Gamma \vdash - \sqsubset -$ on neutral terms, which is a total ordering. We define an auxiliary total ordering $\Gamma \vdash - \sqsubset_\tau -$ on pure normal terms that share the type τ, and give mutually inductive definitions of the two (Figure 8). Since we cannot synthesise neutral terms that contain match terms—a limitation we inherit from Myth, see the main paper—we omit the ordering on those here.

We first define some additional notation. $\Gamma_1 \leq \Gamma$ denotes that the typing context Γ_1 is a prefix of Γ, that is $\Gamma = \Gamma_1, \Gamma_2$ for some Γ_2. $\Gamma \vdash t$ denotes that t is well-defined under Γ, similar to $\Gamma \vdash t : \tau$ but where τ is irrelevant. $\Gamma \vDash t_1, t_2$ denotes that t_1 and t_2 are equally defined under Γ, meaning that for all $\Gamma_1 \leq \Gamma$ we have $\Gamma_1 \vdash t_1 \iff \Gamma_1 \vdash t_2$. $M_1 \sqsubset_d M_2$ is a transitive relation between neutrals based only on the top-level destructor, that is

$$x \sqsubset_d \pi_1 t \sqsubset_d \pi_2 t \sqsubset_d t\, t$$

where x is any variable and each t represents any term.

Both of these relations, with fixed Γ and, if applicable, τ, are irreflexive, antisymmetric, transitive, and total. Additionally, extending a context Γ with a new variable x preserves the ordering on already-definable terms, places all newly-definable terms above them, and gives x as the least of the newly-definable terms. Using these properties, we can replace the rules NFAbs* and NFMatch* with those shown in Figure 9 (changes highlighted), and see that this does not weaken the preconditions.

B.2 Unique Normal Forms

Balat et al.'s construction gives every term a normal form [4, Theorem 5.1]. With our described changes, we can show the stronger condition that every term has a *unique* normal form; rather than building directly on Balat et al.'s theorem, we construct our proof by way of Altenkirch et al.'s normal forms [2].

Theorem 1 (Uniqueness of Normal Forms). *For every STLC+ term $\Gamma \vdash t : \tau$, there exists a unique normal term $\Gamma \vdash_{\mathrm{N}} N : \tau$ such that $\Gamma \vdash t \simeq_\tau N$. Therefore, if $\Gamma \vdash_{\mathrm{N}} N_1, N_2 : \tau$, then $N_1 = N_2 \iff \Gamma \vdash N_1 \simeq_\tau N_2$.*

Proof. Follows from Altenkirch et al.'s results [2] and that their reification function [2, Definition 3.2], when scrutinees are chosen by our ordering, is an injective mapping from their normal forms to ours. □

Corollary 1. *If $\Gamma \vdash_{\mathrm{P}} P_1, P_2 : \tau$, then $P_1 = P_2$ if and only if $\Gamma \vdash P_1 \simeq_\tau P_2$. If $\Gamma \vdash_{\mathrm{M}} M_1, M_2 : \tau$, then $M_1 = M_2$ if and only if $\Gamma \vdash M_1 \simeq_\tau M_2$.*

$$\frac{\Gamma_1 \leq \Gamma \qquad \Gamma_1 \vdash M_1 \qquad \Gamma_1 \not\models M_2}{\Gamma \vdash M_1 \sqsubset M_2}\ \textsc{OrdCtx}$$

$$\frac{\Gamma \vDash M_1, M_2 \qquad M_1 \sqsubset_d M_2}{\Gamma \vdash M_1 \sqsubset M_2}\ \textsc{OrdOp}$$

$$\frac{\Gamma \vdash M_1 \sqsubset M_2}{\Gamma \vdash \pi_1 M_1 \sqsubset \pi_1 M_2}\ \textsc{OrdFst} \qquad \frac{\Gamma \vdash M_1 \sqsubset M_2}{\Gamma \vdash \pi_2 M_1 \sqsubset \pi_2 M_2}\ \textsc{OrdSnd}$$

$$\frac{\Gamma \vDash M_1\, P_1, M_2\, P_2 \qquad \Gamma \vdash M_1 \sqsubset M_2}{\Gamma \vdash M_1\, P_1 \sqsubset M_2\, P_2}\ \textsc{OrdAppFunc}$$

$$\frac{\Gamma \vDash M\, P_1, M\, P_2 \qquad \Gamma \vdash M : \tau_1 \to \tau \qquad \Gamma \vdash P_1 \sqsubset_{\tau_1} P_2}{\Gamma \vdash M\, P_1 \sqsubset M\, P_2}\ \textsc{OrdAppArg}$$

$$\frac{\Gamma_1 \leq \Gamma \qquad \Gamma_1 \vdash P_1 : \tau \qquad \Gamma_1 \not\models P_2}{\Gamma \vdash P_1 \sqsubset_\tau P_2}\ \textsc{OrdCtxPure}$$

$$\frac{\Gamma \vdash M_1, M_2 : \theta_i \qquad \Gamma \vdash M_1 \sqsubset M_2}{\Gamma \vdash M_1 \sqsubset_{\theta_i} M_2}\ \textsc{OrdNeutral}$$

$$\frac{x : \tau_1, \Gamma \vdash P_1 \sqsubset_\tau P_2}{\Gamma \vdash \lambda(x : \tau_1).P_1 \sqsubset_{\tau_1 \to \tau} \lambda(x : \tau_1).P_2}\ \textsc{OrdAbs}$$

$$\frac{\Gamma \vDash \langle P_{11}, P_{12}\rangle, \langle P_{21}, P_{22}\rangle \qquad \Gamma \vdash P_{11} \sqsubset_{\tau_1} P_{21}}{\Gamma \vdash \langle P_{11}, P_{12}\rangle \sqsubset_{\tau_1 \times \tau_2} \langle P_{21}, P_{22}\rangle}\ \textsc{OrdPair1}$$

$$\frac{\Gamma \vDash \langle P, P_{12}\rangle, \langle P, P_{22}\rangle \qquad \Gamma \vdash P_{12} \sqsubset_{\tau_2} P_{22}}{\Gamma \vdash \langle P, P_{12}\rangle \sqsubset_{\tau_1 \times \tau_2} \langle P, P_{22}\rangle}\ \textsc{OrdPair2}$$

$$\frac{\Gamma \vDash P_1, P_2}{\Gamma \vdash \mathtt{in}_L^{\tau_1+\tau_2} P_1 \sqsubset_{\tau_1+\tau_2} \mathtt{in}_R^{\tau_1+\tau_2} P_2}\ \textsc{OrdInjections}$$

$$\frac{\Gamma \vdash P_1 \sqsubset_{\tau_1} P_2}{\Gamma \vdash \mathtt{in}_L^{\tau_1+\tau_2} P_1 \sqsubset_{\tau_1+\tau_2} \mathtt{in}_L^{\tau_1+\tau_2} P_2}\ \textsc{OrdInL}$$

$$\frac{\Gamma \vdash P_1 \sqsubset_{\tau_2} P_2}{\Gamma \vdash \mathtt{in}_R^{\tau_1+\tau_2} P_1 \sqsubset_{\tau_1+\tau_2} \mathtt{in}_R^{\tau_1+\tau_2} P_2}\ \textsc{OrdInR}$$

Fig. 8: Rules for our ordering relations.

$$\frac{\Gamma, x : \tau_1 \vdash_{\mathrm{N}} N : \tau \qquad \boxed{N = \delta(M, \dots) \implies \Gamma, x : \tau_1 \vdash x \sqsubseteq M}}{\Gamma \vdash_{\mathrm{P}} \lambda(x : \tau_1).N : \tau_1 \to \tau}\ \textsc{NFAbs}$$

$$\frac{\begin{array}{c}\Gamma \vdash_{\mathrm{M}} M : \tau_1 + \tau_2 \\ \forall i \in \{1, 2\}.\ \Gamma, x_i : \tau_i \vdash_{\mathrm{N}} N_i : \tau \wedge \\ \boxed{N_i = \delta(M_i, \dots) \implies \Gamma, x : \tau_i \vdash M \sqsubset M_i} \\ x_1 \notin FV(N_1) \wedge x_2 \notin FV(N_2) \implies N_1 \neq N_2\end{array}}{\Gamma \vdash_{\mathrm{N}} \delta(M,\ x_1.N_1,\ x_2.N_2) : \tau}\ \textsc{NFMatch}$$

Fig. 9: Our modified normal form rules, with our ordering on scrutinees.

C NSynC

Since we do not wish to provide concrete functions to our synthesiser as example outputs—this would amount to giving the synthesiser the answer—we extend the definition of values to include *partial functions*. A partial function $\cdot \vdash \tilde{f} : \tau_1 \to \tau$ relates distinct values of type τ_1 (the set of which is written $\mathrm{dom}(\tilde{f})$) to values of type τ. Extensional equality relates partial functions to each other and to concrete functions when they agree on all inputs for which both are defined.

The input specification for our synthesis algorithm is a *typed example set* $X : \mathrm{Ex}[\Gamma; \tau]$, consisting of input-output examples $\sigma \mapsto v$. The input σ is an assignment of values of appropriate type to the variables in Γ. The output is a value $\cdot \vdash v : \tau$ which may only contain partial, not concrete, functions.

We also define *neutral constraints* $c ::= M \sqsubset \mid M \sqsubseteq \mid \top$, and write $\Gamma \vdash c(M')$ to mean that M' satisfies c in context Γ: $\Gamma \vdash (M \sqsubset)(M')$ and $\Gamma \vdash (M \sqsubseteq)(M')$ expand in the obvious way, and $\Gamma \vdash \top(M')$ is always true. Constraints let us place lower bounds on term enumeration, and enforce the ordering on neutrals.

C.1 Overview

Normal forms give us a duplication-free search space for synthesis. No two normal forms have the same semantics, and yet every possible semantics is represented in some normal form, so any term we could have synthesised without normalisation has an equivalent that we can synthesise now.

Since our target language is typed and purely functional, the type-directed synthesis algorithm Myth [7] is a natural choice for synthesising it. Our synthesis begins with a top-down, recursive process of generating constructors based on the goal type and example sets. When we are unable to generate a constructor—either at base type, or when examples include both left and right injections—we use an enumerative search to find a neutral term that solves the current subproblem or a match term that partitions the example set in such a way that synthesis can progress.

The algorithm halts when it finds a satisfying term. To prevent it from enumerating indefinitely, we also impose a limit on the sizes of neutral terms the algorithm can produce. An outer loop then iteratively raises that limit until the top-down search can find a satisfying term, or a time limit is reached.

Our algorithm, with Myth, is defined in terms of synthesis relations which govern how the algorithm's inputs relate to its required output. We present these relations in Subappendix C.2 and the algorithm that arises from them in Subappendix C.3; we also present correctness theorems alongside each formulation.

C.2 Synthesis Relations

The three synthesis relations, corresponding to neutral, pure normal, and normal terms respectively, are written

$$\begin{aligned} [\Gamma;\tau], c \rightsquigarrow_{\mathrm{M}} M \quad & (M\text{-guess, Figure 10}) \\ X : \mathrm{Ex}[\Gamma;\tau] \rightsquigarrow_{\mathrm{P}} P \quad & (P\text{-refine, Figure 11), and} \\ X : \mathrm{Ex}[\Gamma;\tau], c \rightsquigarrow_{\mathrm{N}} N \quad & (N\text{-refine, Figure 12),} \end{aligned}$$

and defined by mutual induction. Each relates a specification to the terms of the appropriate sort that satisfy it, and therefore that we wish the synthesiser to produce.

The M-guess relation enumerates all neutral terms $\Gamma \vdash_{\mathrm{M}} M : \tau$ satisfying c; since neutrals are stuck at free variables, there is no way to use an example set to narrow down the search, so it is omitted from the relation. P-refinement deductively synthesises constructors based on the examples and the goal type, e.g. a goal of product type induces a pair constructor. N-refinement synthesises match terms, with the constraint applied to the scrutinee, and also allows producing a pure neutral if no more matches are needed.

The distinction between P- and N-refinement and the boxed sections of the rules do not appear in the corresponding relations of Myth.

Example 2. To illustrate these rules, consider synthesising a term with the semantics $a \wedge b$, where a and b are Boolean values. We begin with examples

$$X = \left\{ \begin{array}{l} [\,\texttt{True}/a,\ \ \texttt{True}/b\ \ ] \mapsto \texttt{True}, \\ [\,\texttt{True}/a,\ \ \texttt{False}/b\,] \mapsto \texttt{False}, \\ [\,\texttt{False}/a, \texttt{True}/b\ \ ] \mapsto \texttt{False}, \\ [\,\texttt{False}/a, \texttt{False}/b\,] \mapsto \texttt{False} \end{array} \right\}$$

with type $X : \mathrm{Ex}[\Gamma;\tau]$, where $\Gamma = a : \texttt{Bool}, b : \texttt{Bool}$ and $\tau = \texttt{Bool}$. To synthesise a solution, we look for some N such that $X : \mathrm{Ex}[\Gamma;\tau], \top \rightsquigarrow_{\mathrm{N}} N$.

N-refine has two rules: NRefinePure and NRefineMatch. We cannot apply NRefinePure, since there are no rules for P-refine that apply to our example set; the goal type only fits PRefineInL and PRefineInR, and our example outputs contain a mixture of left and right injections. To apply NRefineMatch, we search for neutral terms of sum type satisfying our constraint (which is currently $\top$); there are two such terms, a and b.

$$\frac{x : \tau \in \Gamma \qquad \boxed{\Gamma \vdash c(x)}}{[\Gamma;\tau], \boxed{c} \rightsquigarrow_{\mathrm{M}} x}\ \textsc{MGuessVar}$$

$$\frac{[\Gamma;\tau' \to \tau], \boxed{\top} \rightsquigarrow_{\mathrm{M}} M \qquad \emptyset : \mathrm{Ex}[\Gamma;\tau'] \rightsquigarrow_{\mathrm{P}} P \qquad \boxed{\Gamma \vdash c(M\,P)}}{[\Gamma;\tau], \boxed{c} \rightsquigarrow_{\mathrm{M}} M\,P}\ \textsc{MGuessApp}$$

$$\frac{[\Gamma;\tau \times \tau'], \boxed{\top} \rightsquigarrow_{\mathrm{M}} M \qquad \boxed{\Gamma \vdash c(\pi_1 M)}}{[\Gamma;\tau], \boxed{c} \rightsquigarrow_{\mathrm{M}} \pi_2 M}\ \textsc{MGuessFst}$$

$$\frac{[\Gamma;\tau' \times \tau], \boxed{\top} \rightsquigarrow_{\mathrm{M}} M \qquad \boxed{\Gamma \vdash c(\pi_2 M)}}{[\Gamma;\tau], \boxed{c} \rightsquigarrow_{\mathrm{M}} \pi_2 M}\ \textsc{MGuessSnd}$$

Fig. 10: Enumeration rules for M-guess.

Say we choose a. We then define the two new example sets, X_1 and X_2:

$$X_1 = \left\{ \begin{array}{l} [\,\texttt{True}/a, \texttt{True}/b \quad \langle\rangle/x_1\,] \mapsto \texttt{True}, \\ [\,\texttt{True}/a, \texttt{False}/b\ \langle\rangle/x_1\,] \mapsto \texttt{False} \end{array} \right\}$$
$$X_2 = \left\{ \begin{array}{l} [\,\texttt{False}/a, \texttt{True}/b \quad \langle\rangle/x_2\,] \mapsto \texttt{False}, \\ [\,\texttt{False}/a, \texttt{False}/b\ \langle\rangle/x_2\,] \mapsto \texttt{False} \end{array} \right\}.$$

Neither is empty, so we try to synthesise branches. Starting with the left branch, we want N_1 s.t. $X_1 : \mathrm{Ex}[\Gamma, x_1 : 1; \texttt{Bool}], (a \sqsubset) \rightsquigarrow_{\mathrm{N}} N_1$. Our example outputs contain both left and right injections, so all we have is NRefineMatch; the only scrutinee we can guess is b, since $\Gamma \not\vdash (a \sqsubset)(a)$. We split the examples:

$$X_{11} = \{[\texttt{True}/a, \texttt{True}/b, \langle\rangle/x_1, \langle\rangle/x_{11}] \mapsto \texttt{True}\}$$
$$X_{12} = \{[\texttt{True}/a, \texttt{False}/b, \langle\rangle/x_1, \langle\rangle/x_{12}] \mapsto \texttt{False}\}.$$

Again, neither is empty: we recurse to the left branch. The only example has a left injection as output, so we apply NRefinePure using PRefineInL. That gives us a new example set $\{[\texttt{True}/a, \texttt{True}/b, \langle\rangle/x_1, \langle\rangle/x_{11}] \mapsto \langle\rangle\}$ with a goal type of 1, so we use PRefineUnit to synthesise $\langle\rangle$. Backtracking and doing the same for the right branch with X_{12}, we get $N_1 = \delta(b,\, x_{11}.\texttt{in}_L^{1+1}\langle\rangle,\, x_{12}.\texttt{in}_R^{1+1}\langle\rangle)$ or, sugared, $N_1 = \texttt{if}\ b\ \texttt{then True else False}$.

On the other branch, we see that we can immediately apply NRefinePure through PRefineInR, and get $N_2 = \texttt{in}_R^{1+1}\langle\rangle$. We could not have applied NRefineMatch here, since we can only enumerate $M = b$ and find $N_{21} = N_{22} = \texttt{in}_R^{1+1}\langle\rangle$, breaking the last precondition of NRefineMatch.

Backtracking again gives us the full solution,

$$N = \texttt{if}\ a\ \texttt{then}\ (\texttt{if}\ b\ \texttt{then True else False})\ \texttt{else False}.$$

$$\frac{[\Gamma;\theta_i], \boxed{\top} \rightsquigarrow_{\mathrm{M}} M \qquad \forall \sigma \mapsto v \in X.\ M[\sigma] \to^* v}{X : \mathrm{Ex}[\Gamma;\theta_i] \rightsquigarrow_{\mathrm{P}} M}\ \textsc{PRefineNeutral}$$

$$\frac{}{X : \mathrm{Ex}[\Gamma;1] \rightsquigarrow_{\mathrm{P}} \langle\rangle}\ \textsc{PRefineUnit}$$

$$\frac{\begin{array}{c} X_1 \triangleq \{(\sigma \cdot [v/x]) \mapsto \tilde{f}(v) \mid \sigma \mapsto \tilde{f} \in X,\ v \in \mathrm{dom}(\tilde{f})\} \\ X_1 : \mathrm{Ex}[\Gamma, x : \tau_1; \tau], \boxed{(x \sqsubseteq)} \rightsquigarrow_{\mathrm{N}} N \end{array}}{X : \mathrm{Ex}[\Gamma; \tau_1 \to \tau] \rightsquigarrow_{\mathrm{P}} \lambda(x : \tau_1).N}\ \textsc{PRefineAbs}$$

$$\frac{\begin{array}{c} \{\sigma \mapsto v_1 \mid \sigma \mapsto \langle v_1, v_2\rangle \in X\} : \mathrm{Ex}[\Gamma;\tau_1] \rightsquigarrow_{\mathrm{P}} P_1 \\ \{\sigma \mapsto v_2 \mid \sigma \mapsto \langle v_1, v_2\rangle \in X\} : \mathrm{Ex}[\Gamma;\tau_2] \rightsquigarrow_{\mathrm{P}} P_2 \end{array}}{X : \mathrm{Ex}[\Gamma;\tau_1 \times \tau_2] \rightsquigarrow_{\mathrm{P}} \langle P_1, P_2\rangle}\ \textsc{PRefinePair}$$

$$\frac{\begin{array}{c} \nexists \sigma \mapsto \mathtt{in}_R^{\tau_1+\tau_2} v \in X \\ \{\sigma \mapsto v \mid \sigma \mapsto \mathtt{in}_L^{\tau_1+\tau_2} v \in X\} : \mathrm{Ex}[\Gamma;\tau_1] \rightsquigarrow_{\mathrm{P}} P \end{array}}{X : \mathrm{Ex}[\Gamma;\tau_1+\tau_2] \rightsquigarrow_{\mathrm{P}} \mathtt{in}_L^{\tau_1+\tau_2} P}\ \textsc{PRefineInL}$$

$$\frac{\begin{array}{c} \nexists \sigma \mapsto \mathtt{in}_L^{\tau_1+\tau_2} v \in X \\ \{\sigma \mapsto v \mid \sigma \mapsto \mathtt{in}_R^{\tau_1+\tau_2} v \in X\} : \mathrm{Ex}[\Gamma;\tau_2] \rightsquigarrow_{\mathrm{P}} P \end{array}}{X : \mathrm{Ex}[\Gamma;\tau_1+\tau_2] \rightsquigarrow_{\mathrm{P}} \mathtt{in}_R^{\tau_1+\tau_2} P}\ \textsc{PRefineInR}$$

Fig. 11: Refinement rules for P-refine.

$$\frac{X : \mathrm{Ex}[\Gamma;\tau] \rightsquigarrow_{\mathrm{P}} P}{X : \mathrm{Ex}[\Gamma;\tau], c \rightsquigarrow_{\mathrm{N}} P}\ \textsc{NRefinePure}$$

$$\frac{\begin{array}{c} [\Gamma;\tau_1+\tau_2], \boxed{c} \rightsquigarrow_{\mathrm{M}} M \\ X_1 \triangleq \{\sigma \cdot [v'/x_1] \mapsto v \mid \sigma \mapsto v \in X, M[\sigma] \to^* \mathtt{in}_L^{\tau_1+\tau_2}(v')\} \\ X_2 \triangleq \{\sigma \cdot [v'/x_2] \mapsto v \mid \sigma \mapsto v \in X, M[\sigma] \to^* \mathtt{in}_R^{\tau_1+\tau_2}(v')\} \\ \forall i \in \{1,2\}.\ X_i \neq \emptyset \wedge X_i : \mathrm{Ex}[\Gamma, x_i : \tau_i; \tau], \boxed{(M \sqsubset)} \rightsquigarrow_{\mathrm{N}} N_i \\ \boxed{x_1 \notin FV(N_1) \wedge x_2 \notin FV(N_2) \implies N_1 \neq N_2} \end{array}}{X : \mathrm{Ex}[\Gamma;\tau], \boxed{c} \rightsquigarrow_{\mathrm{N}} \delta(M,\ x_1.N_1,\ x_2.N_2)}\ \textsc{NRefineMatch}$$

Fig. 12: Refinement rules for N-refine.

Correctness. We show that the synthesis relations are sound (Theorem 2), in the sense that they only synthesise terms satisfying the example set, and semantically optimal (Theorem 3), meaning only semantically distinct terms can be synthesised.

Unfortunately, our algorithm is not complete. We inherit from Myth the requirement when synthesising match terms that the subset of examples going to each branch be non-empty. This requirement prevents synthesising arbitrary match terms that do not further the synthesis goal and ensures termination when the goal can be satisfied, but comes at the cost of blocking certain terms: since M-guessing doesn't propagate examples, NSynC can't enumerate neutrals that contain matches as subterms. Since by the grammar such a neutral must be or contain an application term, we call them terms with *branching arguments*. Theorem 4 shows that our synthesis relations are complete for terms without branching arguments.

Myth can also synthesise all of the normal forms that NSynC can, with the same restriction on branching arguments. Therefore, as long as normalisation does not introduce branching arguments—we believe that this holds, though the proof requires reasoning about the normalisation procedure itself—restricting our synthesis to normal terms does not reduce completeness.

Theorem 2 (Soundness). *If* $X : \mathrm{Ex}[\Gamma;\tau], \top \rightsquigarrow_{\mathrm{N}} N$ *then* N *satisfies* X*, that is for all* $\sigma \mapsto v \in X$*,* $N[\sigma] \to^* v$*.*

Proof. In PRefineNeutral, satisfaction of the example set is checked explicitly, and for PRefineUnit it is trivial; that the other refinement rules are sound follows by induction. □

Theorem 3 (Semantic Optimality). *If* $X : \mathrm{Ex}[\Gamma;\tau], c \rightsquigarrow_{\mathrm{N}} N_1$ *and* $X : \mathrm{Ex}[\Gamma;\tau], c \rightsquigarrow_{\mathrm{N}} N_2$*, then* $\Gamma \vdash N_1 \simeq_\tau N_2$ *if and only if* $N_1 = N_2$*, and similarly for* $X : \mathrm{Ex}[\Gamma;\tau] \rightsquigarrow_{\mathrm{P}} P_1, P_2$ *and* $[\Gamma;\tau], c \rightsquigarrow_{\mathrm{M}} M_1, M_2$*.*

Proof. The synthesis relations erase to the normal form rules (Subappendix B), so the terms produced by N-refine are normal (by P-refine are pure normal, and by M-guess are neutral), then by Theorem 1 (and its corollary). □

To prove bounded completeness for N-refine, we first need to show bounded completeness for M-guessing via the following lemma.

Lemma 1. *For any neutral term* $\Gamma \vdash_{\mathrm{M}} M : \tau$ *with no branching arguments, and any constraint* c *s.t.* $\Gamma \vdash c(M)$*,* $[\Gamma;\tau], c \rightsquigarrow_{\mathrm{M}} M$*.*

Proof. All four M-guess rules check the constraint explicitly rather than propagating it, so we can ignore the constraint in the rest of the proof and check that $\Gamma \vdash_{\mathrm{M}} M : \tau$ implies $[\Gamma;\tau], \top \rightsquigarrow_{\mathrm{M}} M$ where M has no branching arguments. This is just induction on the structure of M, with the observation that having an empty example set and no match terms renders the synthesis and normalisation rules almost identical. □

Theorem 4 (Bounded Completeness). *For any term* $\Gamma \vdash t : \tau$ *whose normal form* N *has no branching arguments, there is at least one example set* X *s.t.* $X : \mathrm{Ex}[\Gamma; \tau], \top \rightsquigarrow_{\mathrm{N}} N$.

Proof sketch. We borrow from Altenkirch et al. [2] the notion of a neutral constrained environment $\Gamma|\Xi$, which gives a context Γ a set Ξ of constraints of the form $M = \mathtt{in}_L^{\tau_1+\tau_2} x_1$ or $M = \mathtt{in}_R^{\tau_1+\tau_2} x_2$, where $\Gamma \vdash_{\mathrm{M}} M : \tau_1 + \tau_2$ and $x_i : \tau_i \in \Gamma$.

We can recurse on the structure of N to find a neutral constrained environment for each branch. If $\Gamma|\Xi \vdash \delta(M,\, x_1.N_1,\, x_2.N_2)$, then we get $\Gamma, x_1 : \tau_1|\Xi, M = \mathtt{in}_L^{\tau_1+\tau_2} x_1 \vdash N_1$ and $\Gamma, x_2 : \tau_2|\Xi, M = \mathtt{in}_R^{\tau_1+\tau_2} x_2 \vdash N_2$. Other term-level symbols may duplicate the environment into branches (pairs) or add to the typing context (abstractions) as usual, but do not modify Ξ.

Once each branch's neutral constrained environment has been found, we construct assignments σ to variables in each Γ that satisfy Ξ, that is where for each $M = \mathtt{in}_L^{\tau_1+\tau_2} x_i \in \Xi$ we have $M[\sigma] \to^* \mathtt{in}_L^{\tau_1+\tau_2}(x_i[\sigma])$. By Theorem 1, all non-identical M terms are semantically distinct, and as a consequence of NFMatch we will not see identical M terms in the same Ξ, so all Ξ are satisfiable.

We now have a set of assignments σ for each branch; the rest of the construction of X, where each input is a restriction of some σ to just the variables in the starting context, follows by reversing the refinement rules. That $X : \mathrm{Ex}[\Gamma; \tau], \top \rightsquigarrow_{\mathrm{N}} N$ is by induction on the structure of N, using the fact that every branch has at least one example in X (by our construction) and that every neutral term can be enumerated (by the lemma). □

C.3 Synthesis Algorithm

The algorithm is simply to exhaustively apply the synthesis relations until a solution emerges, or there are no more rules to apply.

Since there are contexts in which M-guessing could relate one input to infinitely many terms, meaning our algorithm would get stuck enumerating them, we bound the size of neutral terms that our synthesiser can produce: n_M bounds the neutrals of base type, and n_δ the neutrals of sum type. With such a limit, synthesis terminates; we then iteratively increase that size limit until a solution is found or a time limit is reached. We also provide a match depth parameter d, with the idea of preventing NSynC from over-specialising its output to the example set by producing too many branches. Since these parameters bound the search space, we also allow the user to specify a procedure for increasing them, so that synthesis attempts can continue until the parameters are large enough for a solution to be found.

The main function of NSynC, shown in Algorithm 1, is just a wrapper around **NRefine** to handle the parameter search. The mutually recursive functions **NRefine** and **PRefine** (Algorithms 2 and 3, respectively) are the key components of NSynC, and implement the N-refine and P-refine relations. **NRefine** tries to apply NRefinePure first, and falls back on **NRefineMatch** if that fails; **PRefine** attempts to apply whichever of the P-refinement rules applies to the input based on its goal type. Both functions return `Fail` if no term could be synthesised within the size bounds.

Algorithm 1: NSynC$(X : \mathrm{Ex}[\Gamma;\tau], (n_M, n_\delta, d), \mathbf{step})$

Data: Example set $X : \mathrm{Ex}[\Gamma;\tau]$, search parameters $n_M, n_\delta, d \in \mathbb{N}$, search parameter increment procedure $\mathbf{step} : \mathbb{N}^3 \to \mathbb{N}^3$
Result: A normal term N with no branching arguments which satisfies X.

$N \leftarrow \mathbf{NRefine}(X : \mathrm{Ex}[\Gamma;\tau]; \top; (n_M, n_\delta, d))$;
while $N = \texttt{Fail}$ **do**
  $n_M, n_\delta, d \leftarrow \mathbf{step}(n_M, n_\delta, d)$;
  $N \leftarrow \mathbf{NRefine}(X : \mathrm{Ex}[\Gamma;\tau]; \top; (n_M, n_\delta, d))$;
end
return N;

Algorithm 2: NRefine$(X : \mathrm{Ex}[\Gamma;\tau]; c; (n_M, n_\delta, d))$

Data: Example set $X : \mathrm{Ex}[\Gamma;\tau]$, constraint c, search parameters $n_M, n_\delta, d \in \mathbb{N}$
Result: A term N with no branching arguments s.t. $X : \mathrm{Ex}[\Gamma;\tau], c \rightsquigarrow_{\mathrm{N}} N$, or $\texttt{Fail}$ if no N exists within the search parameters.

$P \leftarrow \mathbf{PRefine}(X : \mathrm{Ex}[\Gamma;\tau]; (n_M, n_\delta, d))$;
if $P \neq \texttt{Fail}$ **then return** P;
else if $d \geq 1$ **then**
  while $M \leftarrow \boldsymbol{MGuessSum}(\Gamma; c; n_\delta)$ **do**
    $c \leftarrow (M \sqsubset)$;
    $\tau_1 + \tau_2 \leftarrow \mathbf{type}(M)$;
    $X_1 \leftarrow \{\sigma \cdot [v'/x_1] \mapsto v \mid \sigma \mapsto v \in X, M[\sigma] \to^* \mathtt{in}_L^{\tau_1+\tau_2}(v')\}$;
    $X_2 \leftarrow \{\sigma \cdot [v'/x_2] \mapsto v \mid \sigma \mapsto v \in X, M[\sigma] \to^* \mathtt{in}_R^{\tau_1+\tau_2}(v')\}$;
    if $X_1 = \emptyset \vee X_2 = \emptyset$ **then continue**;
    $N_1 \leftarrow \mathbf{NRefine}(X_1 : \mathrm{Ex}[\Gamma, x_1 : \tau_1; \tau]; (M \sqsubset); (n_M, n_\delta; d-1))$;
    if $N_1 = \texttt{Fail}$ **then continue**;
    if $x_1 \notin FV(N_1) \wedge \forall \sigma \mapsto v \in X.\, N_1[\sigma] \to^* v$ **then return** N_1;
    $N_2 \leftarrow \mathbf{NRefine}(X_2 : \mathrm{Ex}[\Gamma, x_2 : \tau_2; \tau]; (M \sqsubset); (n_M, n_\delta; d-1))$;
    if $N_2 = \texttt{Fail}$ **then continue**;
    return $\delta(M,\, x_1.N_1,\, x_2.N_2)$;
  end
end
return $\texttt{Fail}$;

Algorithm 3: PRrefine$(X : \mathrm{Ex}[\Gamma;\tau];(n_M,n_\delta,d))$

Data: Example set $X : \mathrm{Ex}[\Gamma;\tau]$, search parameters $n_M, n_\delta, d \in \mathbb{N}$
Result: A term P with no branching arguments s.t. $X : \mathrm{Ex}[\Gamma;\tau] \rightsquigarrow_{\mathrm{P}} P$, or `Fail` if no P exists within the search parameters.

match τ **with**
  case θ_i **do**
    $c \leftarrow \top$;
    while $M \leftarrow$ ***MGuessBase***$(\Gamma;\theta_i;c;n_M)$ **do**
      if $\forall \sigma \mapsto v \in X.\, M[\sigma] \rightarrow^* v$ **then return** M;
      $c \leftarrow (M \sqsubset)$;
    end
    return `Fail`;
  case 1 **do return** $\langle\rangle$;
  case $\tau_1 \to \tau_2$ **do**
    $X' \leftarrow \{(\sigma \cdot [v/x]) \mapsto \tilde{f}(v) \mid \sigma \mapsto \tilde{f} \in X,\, v \in \mathrm{dom}(\tilde{f})\}$;
    $N \leftarrow$ **NRefine**$(X' : \mathrm{Ex}[\Gamma, x : \tau_1; \tau_2]; (x \sqsubseteq); (n_M, n_\delta, d))$;
    if $N =$ `Fail` **then return** `Fail`;
    return $\lambda(x : \tau_1).N$;
  case $\tau_1 \times \tau_2$ **do**
    $P_1 \leftarrow$ **PRefine**$(\{\sigma \mapsto v_1 \mid \sigma \mapsto \langle v_1, v_2\rangle \in X\} : \mathrm{Ex}[\Gamma;\tau_1]; (n_M,n_\delta,d))$;
    if $P_1 =$ `Fail` **then return** `Fail`;
    $P_2 \leftarrow$ **PRefine**$(\{\sigma \mapsto v_2 \mid \sigma \mapsto \langle v_1, v_2\rangle \in X\} : \mathrm{Ex}[\Gamma;\tau_2]; (n_M,n_\delta,d))$;
    if $P_2 =$ `Fail` **then return** `Fail`;
    return $\langle P_1, P_2\rangle$
  case $\tau_1 + \tau_2$ **do**
    if $\nexists \sigma \mapsto \mathit{in}_R^\tau v \in X$ **then**
      $P \leftarrow$ **PRefine**$(\{\sigma \mapsto v \mid \sigma \mapsto \mathtt{in}_L^\tau(v) \in X\} : \mathrm{Ex}[\Gamma;\tau_1]; (n_M,n_\delta,d)$;
      if $P =$ `Fail` **then return** `Fail`;
      return $\mathit{in}_L^\tau(P)$;
    else if $\nexists \sigma \mapsto \mathit{in}_L^\tau v \in X$ **then**
      $P \leftarrow$ **PRefine**$(\{\sigma \mapsto v \mid \sigma \mapsto \mathtt{in}_R^\tau(v) \in X\} : \mathrm{Ex}[\Gamma;\tau_2]; (n_M,n_\delta,d)$;
      if $P =$ `Fail` **then return** `Fail`;
      return $\mathit{in}_R^\tau(P)$;
    else
      return `Fail`;
    end
  end
end

Both of these algorithms call enumeration functions which implement M-guessing; **MGuessSum** and **MGuessBase**, respectively. Both are wrappers around a successor function $S(\Gamma, M, \tau^{\Diamond}, n_M)$, which gives the least neutral term M' with no branching arguments s.t. $\Gamma \vdash_{\mathrm{N}} M' : \tau^{\Diamond}$, $\Gamma \vdash M \sqsubset M'$, and the size of M' is at most n_M (or `Fail` if there is no such term). $\tau^{\Diamond}$ here is a type template; for example, $\theta_i \times \Diamond$ represents any product type with θ_i as the first element, and we enumerate neutrals of this type template to get $\Gamma \vdash \pi_1 M : \theta_i$. The full definition of the successor function is omitted for space; it follows from the definition of the ordering relation and the M-guessing rules. The definition of **MGuessSum** is

$$\mathbf{MGuessSum}(\Gamma; (M \sqsubset); n_\delta) \triangleq S(\Gamma, M, \Diamond + \Diamond, n_\delta)$$

$$\mathbf{MGuessSum}(\Gamma; (M \sqsubseteq); n_\delta) \triangleq \begin{cases} M & \text{if } \Gamma \vdash M : \Diamond + \Diamond \\ & \quad \wedge \operatorname{size}(M) \leq n_\delta \\ S(\Gamma, M, \Diamond + \Diamond, n_\delta) & \text{otherwise} \end{cases}$$

$$\mathbf{MGuessSum}(\Gamma; \top; n_\delta) \triangleq \mathbf{MGuessSum}(\Gamma; (x_0 \sqsubseteq); n_\delta),$$

where in the last case x_0 refers to the first variable in Γ, and **MGuessSum** returns `Fail` if Γ is empty. **MGuessBase** is definied similarly, but searches for a concrete type θ_i instead of $\Diamond + \Diamond$.

Correctness. As NSynC is a faithful implementation of the synthesis relations, we get identical correctness guarantees as long as the parameter search is well-behaved:

Theorem 5 (Correctness of NSynC). *Assume* ***step*** *and n_M, n_δ, d are such that iterating* ***step*** *on n_M, n_δ, d tends all three to infinity. Then:*

1. ***Soundness.*** *If* $\boldsymbol{NSynC}(X : \mathrm{Ex}[\Gamma; \tau], (n_M, n_\delta, d), \boldsymbol{step}) = N$*, then N satisfies X.*
2. ***Semantic Optimality.*** *If any function of* ***NSynC*** *enumerates a term, it does not enumerate the same term except as part of a different function call.*
3. ***Bounded Completeness.*** *If there are any $\Gamma \vdash_{\mathrm{N}} N : \tau$ with no branching arguments that satisfy X, then there is one s.t.*

$$\boldsymbol{NSynC}(X : \mathrm{Ex}[\Gamma; \tau], (n_M, n_\delta, d), \boldsymbol{step}) = N$$

Proof sketch. We assume (since we omit its definition) correctness of the successor function and take correctness of **MGuessSum** and **MGuessBase** as base cases for our induction. In each case, we can easily check that the manipulation of the neutral constraint is correct; $x_0 \sqsubset$ is equivalent to $\top$ because x_0 is the least neutral term (a free variable by OrdOp, and the first by OrdCtx), and the rest are trivial. This lets us show that iterating **MGuessSum** and **MGuessBase** is equivalent to enumerating the right-hand side of the M-guess relation, subject to the size limit.

We can show by induction that every time **NRefine** or **PRefine** returns a term, this term satisfies the corresponding relation; **MGuessBase** and the unit type case give us base cases for **PRefine**. Soundness follows from Theorem 2. Semantic optimality requires the extra observation that the only terms considered by any function are those returned by recursive calls, and then follows from Theorem 3.

We can also show by induction that any derivation of $X : \mathrm{Ex}[\Gamma; \tau], \top \rightsquigarrow_{\mathrm{N}} N$ corresponds to a trace of NSynC, provided the search parameters are large enough, unless NSynC returns some other term N' first, and in that case soundness implies that $X : \mathrm{Ex}[\Gamma; \tau], \top \rightsquigarrow_{\mathrm{N}} N'$ regardless. Bounded completeness follows from Theorem 4. □

D Evaluation

We implement NSynC—and re-implement Myth for comparison—in Haskell with GHC 9.6.7. We conduct our experiments on a 10-core 2.80 GHz Intel i9-10900 with 128GB of RAM, running Ubuntu 22.04.

We generate synthetic benchmarks by randomly generating, in order, a typing context, a goal type inhabitable in that context, a term of that type, and a set of examples for that term. For our preliminary tests, we use four base types, each of which has up to ten distinct values, and produce a typing context of six variables. Each type in the context has a maximum depth of four, and the goal type has a maximum depth of six; we also restrict the goal type to only include the base types that appear in the typing context, otherwise the goal type will be uninhabitable. Our synthesised terms have a maximum neutral size of 10, and a maximum match depth of 5. For each term, we produce 20 examples, and values that are partial functions are generated with six input-output pairs.

Under these parameters, we attempted to generate 200 benchmarks, of which 166 were generated successfully. On the other 32, our generator timed out before finding a term that inhabited the goal type. Myth*, our re-implementation, was able to solve 141 of the benchmarks, and NSynC solved 104.

We use Haskell's `criterion` package to time Myth* and NSynC on each benchmark, with a time limit of 60 seconds for each run. Additionally, we compute the total number of scrutinees enumerated by each algorithm in solving each benchmark. By measuring scrutinees, we can see the number of match terms checked by each algorithm, which we expect to be the majority of the work that NSynC eliminates.

Results are presented in the main paper.