Programming by Navigation

JUSTIN LUBIN, University of California, Berkeley, USA
PARKER ZIEGLER, University of California, Berkeley, USA
SARAH E. CHASINS, University of California, Berkeley, USA

When a program synthesis task starts from an ambiguous specification, the synthesis process often involves an iterative specification refinement process. We introduce the Programming by Navigation Synthesis Problem, a new synthesis problem adapted specifically for supporting iterative specification refinement in order to find a *particular* target solution. In contrast to prior work, we prove that synthesizers that solve the Programming by Navigation Synthesis Problem show *all* valid next steps (**Strong Completeness**) and *only* valid next steps (**Strong Soundness**). To meet the demands of the Programming by Navigation Synthesis Problem, we introduce an algorithm to turn a type inhabitation oracle (in the style of classical logic) into a fully constructive program synthesizer. We then define such an oracle via sound compilation to Datalog. Our empirical evaluation shows that this technique results in an efficient Programming by Navigation synthesizer that solves tasks that are either impossible or too large for baselines to solve. Our synthesizer is the first to guarantee that its specification refinement process satisfies both **Strong Completeness** and **Strong Soundness**.

CCS Concepts: • **Software and its engineering** → **Automatic programming**.

Additional Key Words and Phrases: Interactive Program Synthesis, Component-Based Synthesis, Datalog

ACM Reference Format:
Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Programming by Navigation. *Proc. ACM Program. Lang.* 9, PLDI, Article 165 (June 2025), 28 pages. https://doi.org/10.1145/3729264

1 Introduction

Program synthesis tasks often begin with an underspecification of a target program [38]. If we care about refining this underspecification to reach not just any program but a *particular* program, then program synthesizers can employ an iterative specification refinement process [55, 83]. Our work starts from the observation that no existing technique for specification refinement offers what we will call **Strong Completeness** and **Strong Soundness**; that is, the guarantee that, at each round of synthesis, the synthesizer presents *all* the valid next steps (**Strong Completeness**) and *only* the valid next steps (**Strong Soundness**).

We therefore introduce the Programming by Navigation Synthesis Problem, an interactive program synthesis problem formulated such that synthesizers solving the problem are guaranteed to provide **Strong Completeness** and **Strong Soundness**. In Programming by Navigation, a step provider takes a work-in-progress program and returns the set of all and only the valid next steps. In contrast to prior work, any step from this set may be selected, and none will refine the specification to be unsatisfiable.

Taken together, **Strong Completeness** and **Strong Soundness** seem to imply that a synthesizer that solves the Programming by Navigation Synthesis Problem must somehow be

Authors' Contact Information: Justin Lubin, University of California, Berkeley, Berkeley, USA, justinlubin@berkeley.edu; Parker Ziegler, University of California, Berkeley, Berkeley, USA, peziegler@cs.berkeley.edu; Sarah E. Chasins, University of California, Berkeley, Berkeley, USA, schasins@cs.berkeley.edu.

ACM 2475-1421/2025/6-ART165
https://doi.org/10.1145/3729264

aware of all possible solutions to the synthesis problem ahead of time, so that the correct paths can be returned. However, enumerating all solutions does not scale when there are many possible solutions to the underspecification and cannot succeed when there are infinitely many solutions.

To meet the demands of the Programming by Navigation Synthesis Problem, we observe that we can transform an *inhabitation oracle*—a procedure that simply returns "yes" or "no" as to whether or not a type is inhabited, in the style of classical logic—into a fully constructive program synthesizer. With the flexibility granted by only needing to return "yes" or "no," we can then define a fast Datalog-backed inhabitation oracle that uses fact derivability to prove type inhabitation.

We implement our solution to the Programming by Navigation Synthesis Problem in a system called Honeybee and evaluate it on a benchmark suite of 21 programs from three domains: generic synthesis tasks, bioinformatics workflows, and geospatial data analyses. We find that, unlike enumeration, Honeybee applies to benchmarks that have infinitely many solutions, and scales well to benchmarks with a finite but large number of solutions.

Contributions. In summary, we contribute:

(1) **Programming by Navigation**: A novel program synthesis problem, formulated such that any synthesizer that solves the problem achieves both **Strong Completeness** and **Strong Soundness** (that is, presents *all* valid next steps and *only* valid next steps).
(2) **Top–Down Classical–Constructive Synthesis**, a solution to the Programming by Navigation Synthesis Problem. We (i) instantiate Programming by Navigation for top-down program construction steps, (ii) introduce an algorithm to solve the Programming by Navigation Synthesis Problem in this setting by transforming a classical-style inhabitation oracle into a constructive program synthesizer, and (iii) provide such an oracle via compilation to Datalog.
(3) **Implementation and Evaluation**: We implement our instantiation of Programming by Navigation in a tool called Honeybee and evaluate its performance empirically.

2 Overview

To illustrate the core insights of our approach, we consider a relatively small example.

Setup. Suppose we define the following structs:

$$\texttt{struct}\ A\langle\mathsf{int}\rangle\ \{\cdots\} \qquad \texttt{struct}\ R\langle\mathsf{int}\rangle\ \{\cdots\} \qquad \texttt{struct}\ M\langle\mathsf{int},\mathsf{int},\mathsf{bool}\rangle\ \{\cdots\} \qquad \texttt{struct}\ D\langle\mathsf{int},\mathsf{int}\rangle\ \{\cdots\}$$

Our goal will be to synthesize a composition of components between these types. These components will be functions annotated with validity conditions for when they are valid to call, but we will not be inspecting the bodies of the functions. Thus, we need not inspect the bodies of these types.

These types are standard product types that one would expect from a mainstream language, with one additional feature. We parameterize each type by zero or more base values like integers or booleans; we call these values the type's *metadata*. For example,[1] in a bioinformatics context, $R\langle 2\rangle$ could be the type of an integer vector where each entry is the number of times a gene transcript appears in a tissue labeled Sample 2. In a geospatial analysis context, $R\langle 90\rangle$ could be the type of a raster image with a spatial resolution of 90 meters^2/pixel. Although the underlying *data* is extremely different in these contexts, we can capture both with the same type-level *metadata*.

Let us now consider some functions over these types:

$$\begin{array}{lll} q_1, q_2 : () \rightarrow_{\varphi_1} R & s : A \rightarrow_{\varphi_2} R & b : M \rightarrow_{\varphi_4} M \\ a_1, a_2 : () \rightarrow_{\varphi_1} A & c : R \times R \rightarrow_{\varphi_3} M & d : M \rightarrow_{\varphi_5} D \end{array}$$

[1]We include these concrete examples to provide intuition but emphasize that our synthesis problem and solution are domain-agnostic. Understanding the domain-specific content is entirely optional for understanding our contributions.

These functions are standard as well, with one additional feature. Each function is annotated with a *validity condition* φ that constrains when it may be called. Importantly, these validity conditions can reference the metadata of their input and output types. For example, these functions may be annotated with the following validity conditions:

$$\varphi_1 := S(\mathsf{ret}_1) \qquad \varphi_3 := \mathsf{ret}_1 = \mathsf{param}_{1,1} \wedge \mathsf{ret}_2 = \mathsf{param}_{2,1} \wedge \mathsf{ret}_3 = \bot$$
$$\varphi_2 := \mathsf{ret}_1 = \mathsf{param}_{1,1} \qquad \varphi_4 := \mathsf{ret}_1 = \mathsf{param}_{1,1} \wedge \mathsf{ret}_2 = \mathsf{param}_{1,2} \wedge \mathsf{ret}_3 = \top \wedge \mathsf{param}_{1,3} = \top$$
$$\varphi_5 := \mathsf{ret}_1 = \mathsf{param}_{1,1} \wedge \mathsf{ret}_2 = \mathsf{param}_{1,2}$$

Validity condition φ_5 for d states that the first metadata argument of the return type (ret_1) must be equal to the first metadata argument of the type of the function's first argument ($\mathsf{param}_{1,1}$), and similarly for the second metadata argument of the return type and the second metadata argument of the function's first argument. Thus, if d is called on an argument of type, say, $M\langle 1, 2, \top\rangle$, then it can have a return type of $D\langle 1, 2\rangle$ but not $D\langle 5, 6\rangle$. Throughout the paper, we use superscripts on a function to refer to the metadata values of a function's return type; for example, $c^{1,2,\top}$ would have return type $M\langle 1, 2, \top\rangle$ and $d^{1,2}$ would have return type $D\langle 1, 2\rangle$. As a concrete example, a function $f : R \rightarrow_{\varphi} R$ with $\varphi := \mathsf{param}_{1,1} < \mathsf{ret}_1$ could be the type of a downsampling operation in geospatial analysis, in which the output raster image must have a less precise spatial resolution than the input.

Validity condition φ_1 demonstrates an important aspect of our predicate language: the ability to query *atomic propositions* like S. In addition to a goal type like $D\langle 1, 2\rangle$, our initial specification will include a set Δ of propositions assumed to be true. For example, $\Delta = \{S(1), S(2)\}$ would mean that S holds (only) on 1 and 2, and φ_1 would hold when a return type's first metadata argument is 1 or 2.

To summarize, we will consider the library of types and functions to be fixed, and our initial specification will consist of the following two pieces of information:

(1) A goal type (e.g., $D\langle 1, 2\rangle$).
(2) A set of atomic propositions to set to true (e.g., $\{S(1), S(2)\}$).

A composition of functions is said to satisfy this specification if it can be assigned the goal type and all validity conditions of the called functions are met given the assumed atomic propositions.

Possible Tasks. We can now, at a high level, consider three different types of synthesis problem:

(1) **The Any task: Return *any* solution that satisfies the specification.** This is the standard program synthesis problem.
(2) **The All task: Return *all* solutions that satisfy the specification.** Some program synthesizers that solve the Any task can be extended to solve the All task by continuing search beyond just the first solution found to exhaustively search the space.
(3) **The Particular task: Return a *particular* solution that satisfies the specification.** When there are finitely many solutions, synthesizers that solve the All task can solve the Particular task by returning all solutions. Another approach is to solicit specification refinements.

In this paper, we turn our attention to the last of these three tasks. Returning to our example, one solution of minimal size is $d^{1,2}(c^{1,2,\bot}(q_1^1(), q_1^2()))$. It turns out that there are 32 solutions in total:

$d^{1,2}(c^{1,2,\bot}(q_1^1(), q_1^2()))$	$d^{1,2}(c^{1,2,\bot}(q_1^1(), s^2(a_1^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(q_1^1(), q_1^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(q_1^1(), s^2(a_1^2()))))$
$d^{1,2}(c^{1,2,\bot}(q_1^1(), q_2^2()))$	$d^{1,2}(c^{1,2,\bot}(q_1^1(), s^2(a_2^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(q_1^1(), q_2^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(q_1^1(), s^2(a_2^2()))))$
$d^{1,2}(c^{1,2,\bot}(q_2^1(), q_1^2()))$	$d^{1,2}(c^{1,2,\bot}(q_2^1(), s^2(a_1^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(q_2^1(), q_1^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(q_2^1(), s^2(a_1^2()))))$
$d^{1,2}(c^{1,2,\bot}(q_2^1(), q_2^2()))$	$d^{1,2}(c^{1,2,\bot}(q_2^1(), s^2(a_2^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(q_2^1(), q_2^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(q_2^1(), s^2(a_2^2()))))$
$d^{1,2}(c^{1,2,\bot}(s^1(a_1^1()), q_1^2()))$	$d^{1,2}(c^{1,2,\bot}(s^1(a_1^1()), s^2(a_1^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(s^1(a_1^1()), q_1^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(s^1(a_1^1()), s^2(a_1^2()))))$
$d^{1,2}(c^{1,2,\bot}(s^1(a_1^1()), q_2^2()))$	$d^{1,2}(c^{1,2,\bot}(s^1(a_1^1()), s^2(a_2^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(s^1(a_1^1()), q_2^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(s^1(a_1^1()), s^2(a_2^2()))))$
$d^{1,2}(c^{1,2,\bot}(s^1(a_2^1()), q_1^2()))$	$d^{1,2}(c^{1,2,\bot}(s^1(a_2^1()), s^2(a_1^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(s^1(a_2^1()), q_1^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(s^1(a_2^1()), s^2(a_1^2()))))$
$d^{1,2}(c^{1,2,\bot}(s^1(a_2^1()), q_2^2()))$	$d^{1,2}(c^{1,2,\bot}(s^1(a_2^1()), s^2(a_2^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(s^1(a_2^1()), q_2^2())))$	$d^{1,2}(b^{1,2,\top}(c^{1,2,\top}(s^1(a_2^1()), s^2(a_2^2()))))$

Even with a precise logical specification, there may be many valid solutions to choose between. For example, bubble sort, insertion sort, merge sort, quick sort, and heap sort (each of which may be stable or unstable) are all sorting functions; the choice to use one or another is context-dependent.

The question, then, is how to have a synthesis approach that enables choosing a particular solution among many. As mentioned above, one option is simply to return *all* solutions, but this only works if there are finitely many solutions. Moreover, when the number of solutions is finite but large, such an approach wastes synthesis time exploring many solutions that are *not* the particular target solution. As we will see in our evaluation (Section 7.2), this approach does not scale when there are combinatorially-many solutions. If we set out to design a program synthesizer around solving the Particular task from the start, can we reach the particular solution more quickly?

ASIDE: What are these types, atomic propositions, and functions, anyway? Throughout Section 2, we keep the working example self-contained and generic to demonstrate our key insights. However, the types and functions we define correspond precisely to a common real-world task that biologists often need to perform in the course of analyzing wet-lab data: computing sets of differentially-expressed genes between two biological samples.

The atomic propositions correspond to what data they have on hand. In the case of differential gene expression, they have raw RNA sequencing data: $S(1)$ and $S(2)$ mean that RNA sequencing data for Samples 1 and 2 are available.

The types correspond to standard datatypes from bioinformatics:

- $A\langle n \rangle$ is an alignment of reads from Sample n to a genome (e.g. in BAM file format).
- $R\langle n \rangle$ is a vector of gene read counts for Sample n.
- $M\langle n_1, n_2, b \rangle$ is a read count matrix for Samples n_1 and n_2, where b is a boolean value to track whether or not the matrix has undergone batch correction to account for batch-level differences in sequencing runs.
- $D\langle n_1, n_2 \rangle$ is the result of a differential gene expression test between Samples n_1 and n_2.

The functions correspond to standard bioinformatics algorithms:

- q_1 and q_2 correspond to transcript quantifiers such as `kallisto` [20] and `salmon` [81].
- a_1 and a_2 correspond to aligners such as `bowtie2` [54] and `STAR` [27].
- s corresponds to a read summarizer such as `featureCounts` [59].
- c (uninterestingly) combines two read count vectors into a matrix.
- b corresponds to a batch corrector such as ComBat-seq [108].
- d corresponds to a differential gene expression test such as DESeq2 [61].

Although Programming by Navigation supports synthesis beyond component-based approaches, our specific instantiation does not inspect function bodies (nor aim to synthesize them). Therefore, the implementations of these bioinformatics algorithms can be off-the-shelf, arbitrarily complex, and in many different languages.

The validity conditions ensure these functions are only called when appropriate. For example, φ_1 ensures raw sequencing data is available to perform quantification or alignment, φ_4 ensures batch correction can only be performed once, and φ_5 states that we can test any read count matrix for differential gene expression, regardless of whether or not it has been batch corrected.

In all, the 32 different programs correspond to different valid combinations of these bioinformatic analyses, each of which may be of interest to a biologist analyzing their data depending on the context. Of particular note is whether or not to use batch correction for a given analysis—a highly contextual and often-contested decision among biologists [77].

Key Insight 1 (§3). Our first key insight is to design a synthesis problem specifically adapted to the Particular task. Starting from a blank program, the synthesizer is required to present a set of valid *steps* that indicate next possible steps to extend the program, where a valid step is one

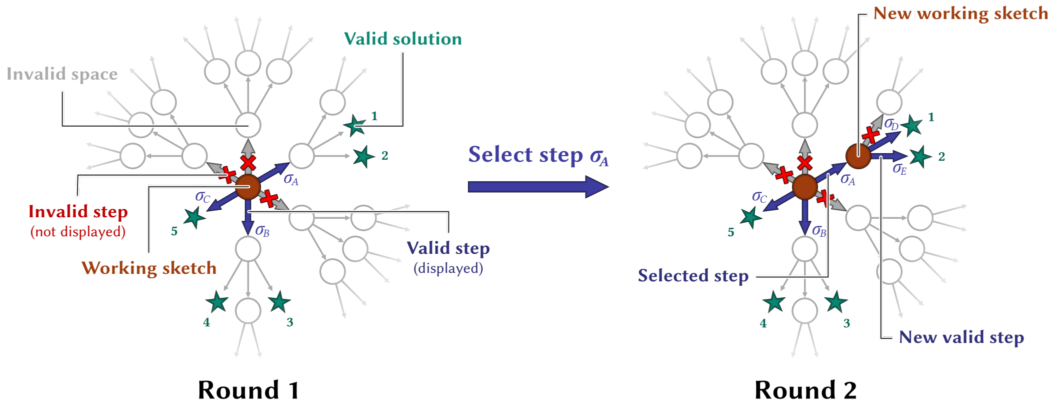

Fig. 1. **Two rounds of a Programming by Navigation. Round 1:** At each round of synthesis, we have a working sketch (i.e., an in-progress program, depicted as an orange circle), which is initially empty. In this scenario, there are 5 valid synthesis solutions (depicted as green stars). Given a working sketch, a Programming by Navigation synthesizer is required to return all steps on paths that lead to valid solutions (depicted as purple arrows annotated with σ_A, σ_B, and σ_C), a property we call **Strong Completeness**. In addition, the synthesizer must not return any steps that cannot lead to a valid solution (depicted as arrows with red crosses), a property we call **Strong Soundness**. **Round 2:** We depict the result of selecting the particular step σ_A (although the others would have been valid to select as well). The step σ_A gets applied to the previous working sketch, resulting in a new working sketch, and the Programming by Navigation synthesizer must again return all and only the valid next steps; now σ_D leads to Solution 1 and σ_E leads to Solution 2.

that can lead to a valid solution. **Crucially, we require that this set of steps can lead to *all* valid solutions, and that it contains *only* steps that lead to a valid solution.** We call the requirement that all valid steps be presented **Strong Completeness** and the requirement that only valid steps be presented **Strong Soundness**.[2] **Strong Completeness** enforces that every valid program is constructible and that each provided set of steps does not eliminate any potential valid programs. **Strong Soundness** enforces that all choices at every intermediate stage of the process lead to a valid solution; synthesis can thus never go down a "rabbit hole" path that leads to no valid solution. The fact that these two guarantees must hold at each round of interaction is the key difference between our problem statement and that of other interactive program synthesizers, as we discuss in Section 10.1.

After the set of all and only valid steps are provided, one of them can be selected to advance to the next round of synthesis. This process repeats until reaching a valid solution. Figure 1 visually depicts two rounds of this process.

We also provide a concrete example of a complete interaction in Figure 2, which shows the steps to reach the 29th solution listed above using "top-down" steps that construct a program from its root node. The table can be read row-by-row, going from left to right in each row. We start in the Working Sketch column, which indicates the in-progress program under construction, using holes ($?_h$) as placeholders to be filled. The Goal column selects one of the holes in the sketch to expand. The synthesizer is then obligated to return precisely the possible fillings in the Options column and no others; these represent the exact set of steps that can lead to a valid solution. Simply exploring

[2]Traditionally, a synthesizer is *sound* if any program it returns satisfies the specification and *complete* if it always returns a program when at least one satisfying solution exists.

WORKING SKETCH	GOAL	OPTIONS	CHOICE
$?_1$	$?_1 : D$	$d^{1,2}(?_2)$	$d^{1,2}(?_2)$
$d^{1,2}(?_2)$	$?_2 : M$	$c^{1,2,\bot}(?_3, ?_4),\ b^{1,2,\top}(?_3)$	$b^{1,2,\top}(?_3)$
$d^{1,2}(b^{1,2,\top}(?_3))$	$?_3 : M$	$c^{1,2,\bot}(?_4, ?_5)$	$c^{1,2,\bot}(?_4, ?_5)$
$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(?_4, ?_5)))$	$?_4 : R$	$q_1^1(),\ q_2^1(),\ q^1(?_6)$	$q^1(?_6)$
$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(q^1(?_6), ?_5)))$	$?_6 : A$	$a_1^1(),\ a_2^1()$	$a_1^1()$
$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(q^1(a_1^1()), ?_5)))$	$?_5 : R$	$q_1^2(),\ q_2^2(),\ q^2(?_6)$	$q^2(?_6)$
$d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(q^1(a_1^1()), q^2(?_6))))$	$?_6 : R$	$a_1^2(),\ a_2^2()$	$a_1^2()$

Solution: $d^{1,2}(b^{1,2,\top}(c^{1,2,\bot}(q^1(a_1^1()), q^2(a_1^2()))))$

Fig. 2. **A step-by-step breakdown of Programming by Navigation for the goal type $D\langle 1, 2\rangle$ and atomic proposition set $\{S(1), S(2)\}$.** Each row is one round of synthesis and is read left-to-right. In each round, we have a working sketch, a goal, and a set of options, which represent all and only the valid next steps. Although all returned options are acceptable to select, the choice in the final column results in the working sketch at the start of the next row. The solution at the bottom is the resulting concrete program after these 7 rounds.

the underlying grammar in a type-directed manner is not sufficient to meet this requirement, as the validity conditions on each function must be met; the options presented must be exactly those with extensions whose validity conditions are all met. Finally, the CHOICE column indicates a choice among these options, which then gets substituted into the working sketch on the next row.

We now have a new question: How can we compute what options are possible at each step of the above interaction? One immediate solution would be to compute the 32 solutions from before and store them in a generalized trie data structure [23], but this solution brings us back to exactly the problem we were trying to avoid: enumerating all solutions is too costly and is in fact impossible in the case of infinitely many solutions.

Key Insight 2 (§4). We observe that, at each intermediate stage, we do not actually need to construct all possible valid solutions. It is enough if we are able to determine if a particular type is inhabited without knowing what that inhabitant actually is. For example, in the second row of Figure 2, we can check if $b^{1,2,\top}(?_3)$ should be included in the set of options by checking if the type of its argument is inhabited; we do not actually need to know what to map $?_3$ to. We formalize this notion in terms of an *inhabitation oracle* that, given a hole and a possible function f to apply, will determine if the type is inhabited by some expression whose root is f.[3] Our inhabitation oracle therefore gives us a "yes" or "no" answer—in the style of classical logic—as to whether or not a type is inhabited. Our second key insight, then, is that we can turn such a classical-style oracle into a synthesizer that actually gives us a term inhabiting a type, in the style of constructive logic.

But one final question remains: How can we define such an inhabitation oracle? One such oracle could be a constructive synthesizer in disguise: simply run a complete ANY program synthesizer on the type, and return "yes" if and only if a solution is found. But this does more work than we require; it actually constructs a solution when one is not needed.[4] Is there some way we can exploit the leeway given by only needing to return a "yes" or "no" answer to design a faster oracle?

Key Insight 3 (§5). Our final key insight is that we can compile library types into Datalog facts and library functions into Datalog rules in such a way that a library type is inhabited if and only if the corresponding fact is derivable. Further, using a variant of cut elimination [36], we can check

[3]Our type inhabitation oracles are unrelated to input-output oracles in the sense of oracle-guided inductive synthesis (OGIS) [46]; in OGIS, oracles are used to provide desired outputs to given program inputs for input-output examples.
[4]We, in fact, implemented this approach and it indeed performed poorly; see Appendix A in the supplementary materials.

whether a library type is inhabited with an expression whose root is a particular function, as is needed for our type inhabitation oracle.[5]

For example, we compile the a_1, b, c, and d functions into the following three rules:

$$\frac{S(x)}{A(x)}\ (\textsc{Rule}_{a_1}) \qquad \frac{M(x,y,\bot)}{M(x,y,\top)}\ (\textsc{Rule}_{b}) \qquad \frac{R(x) \quad R(y)}{M(x,y,\bot)}\ (\textsc{Rule}_{c}) \qquad \frac{M(x,y,b)}{D(x,y)}\ (\textsc{Rule}_{d})$$

The facts of this Datalog program are exactly the types from our library, and a type is inhabited if and only if the corresponding fact is derivable. Further, a derivation of a fact can end in $\textsc{Rule}_f$ if and only if the corresponding type is inhabited by a function application with f at its head.

Our inhabitation oracle will be given a hole name h and a possible function f and return the values $v_1, \dots, v_N$ for which we can replace $?_h$ by some expression starting with $f^{v_1,\dots,v_N}$. We can use the above Datalog facts and rules to do so by constructing a query rule that holds precisely on such $v_1, \dots, v_N$. For example, given an expression $d^{1,2}(?_2)$ with $?_2 : M$, we can check what values x, y, b of metadata are inhabited by a ground program at $?_2$ with the following query based on $\textsc{Rule}_d$:

$$\frac{M(x,y,b) \quad x = 1 \quad y = 2}{Q^\star(x,y,b)}\ (\textsc{Query})$$

However, this query is not enough; we specifically need to see which functions we can use as options in our Programming by Navigation interaction. So, we can specifically check if $?_2$ can be inhabited with b or c functions by *cutting* the corresponding rules ($\textsc{Rule}_b$ and $\textsc{Rule}_c$) in:

$$\frac{M(x,y,\bot) \quad x = 1 \quad y = 2 \quad b = \top}{Q^\star(x,y,b)}\ (\textsc{Rule}_b/\textsc{Query}) \qquad \frac{R(x) \quad R(y) \quad x = 1 \quad y = 2 \quad b = \bot}{Q^\star(x,y,b)}\ (\textsc{Rule}_c/\textsc{Query})$$

The rules $\textsc{Rule}_b$ and $\textsc{Rule}_c$ have essentially been inlined (i.e., cut), so if the resulting queries are satisfiable, we know that we can construct an equivalent derivation tree where we do use those rules. These are the queries that we will use for our type inhabitation oracle.

In summary, our Datalog-backed type inhabitation oracle works by compiling the library to Datalog and checking the derivability of these cut queries.[6]

Summary. Overall, we design a novel synthesis problem that directly solves the PARTICULAR task (Section 3). We then introduce an algorithm that transforms an inhabitation oracle into a synthesizer that solves this problem (Section 4). Finally, we use Datalog to efficiently implement an oracle in a way that does not require that we construct the solutions ahead of time (Section 5).

3 Programming by Navigation

We now introduce the general form of the Programming by Navigation framework. We will begin by specifying what properties steps must satisfy (Section 3.1) and giving a concrete example of such steps in the form of top-down program construction (Section 3.1.1). Next, we will state the Programming by Navigation Synthesis Problem, which—in contrast to prior interactive synthesis problems—requires that *all* and *only* the valid next steps be shown. In Section 3.3, we will show that, in addition to providing all and only the valid steps at each round of the interactive process, solutions to the Programming by Navigation Synthesis Problem (i) terminate immediately if there is no solution, (ii) always progress until a valid solution is found, and (iii) enable all valid expressions to be constructed using only the provided steps.

[5]Throughout this paper, we use "cut" in the sense of fusing proofs, not in the sense of stopping backtracking as in Prolog.
[6]From this perspective, the process in Figure 2 can be viewed as interactively constructing the *provenance* (i.e., derivation tree) of our goal fact. By associating facts with types, we can leverage the Curry-Howard correspondence so that proof trees of these facts correspond to programs satisfying our specification.

$$\textbf{Hole names } h, \quad \textbf{Functions } f \qquad \textbf{Steps } \sigma ::= ?_h \rightsquigarrow f(e_1, \ldots, e_N) \mid \sigma_1 \,;\, \sigma_2$$

$$\textbf{Expressions } e ::= f(e_1, \ldots, e_N) \mid ?_h \qquad \textbf{Step sets } \Sigma ::= \{\sigma_1, \ldots, \sigma_N\} \ \textit{(finite, unordered)}$$

$\boxed{e_1 \xrightarrow{\sigma} e_2}$ σ top-down steps e_1 to e_2

Step/Extend

$$\frac{\mathsf{arity}(f) = N \qquad ?_h \lessdot e}{e \xrightarrow{?_h \rightsquigarrow f(e_1,\ldots,e_N)} [?_h \mapsto f(e_1, \ldots, e_N)]e}$$

Step/Seq

$$\frac{e \xrightarrow{\sigma_1} e' \qquad e' \xrightarrow{\sigma_2} e''}{e \xrightarrow{\sigma_1 \,;\, \sigma_2} e''}$$

Fig. 3. **A Concrete Instantiation of Programming by Navigation.** We assume that each function f has a fixed arity $\mathsf{arity}(f)$ and write $e_1 \lessdot e_2$ if e_1 is a subterm of e_2. These steps model a top-down program construction process (Step/Extend replaces a hole with a function application and Step/Seq sequences steps), but other sets of steps that satisfy the properties of a navigation relation (Definition 3.1) are equally valid.

3.1 Expressions and Steps

Programming by Navigation requires a notion of expressions e and steps σ as well as a relation $e_1 \xrightarrow{\sigma} e_2$ that holds if σ transforms e_1 to e_2. Additionally, it requires a notion of validity on expressions (notated e valid); this validity could be satisfying input-output examples, logical specifications, type checking, or something else entirely. The following central definition captures precisely what kinds of expressions and steps are permissible for Programming by Navigation.

Definition 3.1. Suppose e valid is a validity condition on expressions. Let $e_1 \xrightarrow{\sigma} e_2$ be a relation on expressions e_1, e_2 and steps σ, and let $e_1 \prec e_2$ hold if and only if $e_1 \xrightarrow{\sigma} e_2$ for some σ. Then $e_1 \xrightarrow{\sigma} e_2$ is a **navigation relation** if it satisfies the following four properties.

(1) **Determinism**: There is at most one e' for each e and σ such that $e \xrightarrow{\sigma} e'$.
(2) **No Loops**: $\prec$ is a strict partial order.
(3) **Reachability**: There exists a lower bound e_{start} on the set of valid expressions.
(4) **Finite Between**: Every infinite ascending chain $e_0 \prec e_1 \prec \cdots$ is unbounded.[7]

We write $e \vdash \sigma$ if there exists e' such that $e \xrightarrow{\sigma} e'$. When $e \vdash \sigma$, we define σe to be the unique such e'. We write $e \vdash \Sigma$ for a step set Σ if $e \vdash \sigma$ for all $\sigma \in \Sigma$.

Determinism ensures that steps are deterministic. **No Loops** ensures that a sequence of steps will never loop back to a previously-explored expression (including by having no effect on an expression). **Reachability** ensures that a notion of a "blank program" (e_{start}) exists from which every valid expression is reachable via some sequence of steps. Finally, **Finite Between** ensures that it is impossible to take infinitely many steps between any two expressions. Subject to these constraints, any notion of expressions and steps will suffice.

3.1.1 A Concrete Instantiation. In Figure 3, we provide one possible choice of expressions and steps—modeling a top-down approach to program construction—that we will use later in the paper as an instantiation of the Programming by Navigation framework. With these steps, we could, for example, model the following program construction sequence:

$$?_0 \xrightarrow{?_0 \rightsquigarrow p(q(),?_1)} p(q(), ?_1) \xrightarrow{?_1 \rightsquigarrow r(?_2)\,;\,?_2 \rightsquigarrow s()} p(q(), r(s())).$$

[7]Put another way, $\prec$ is *Noetherian* (satisfies the ascending chain condition) on the principal ideal of e for all expressions e.

We take $e_{\text{start}} = ?_0$ and require that e valid not hold on incomplete programs (programs with holes). We also require hole names to be unique in expressions. The following theorem establishes that these steps are valid to use in the Programming by Navigation framework.[8]

Theorem 3.2. *The top-down step relation is a navigation relation.*

We emphasize that these are not the only possible steps to use; the remainder of this section applies to any notion of steps that satisfies the properties in Definition 3.1. Other kinds of steps—such as those capable of capturing a bottom-up or middle-out process of construction—would be equally valid to slot into this framework if a suitable analogue of Theorem 3.2 can be proven.

3.2 Programming by Navigation Synthesis Problem

With a proper notion of steps established, we can now formalize the exact requirements of a Programming by Navigation synthesizer. We model Programming by Navigation synthesizers as step providers, defined as follows.

Definition 3.3. A **step provider** $\mathbb{S}$ maps expressions to step sets. An $\mathbb{S}$**-interaction** is a finite sequence $e_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_N} e_N$ such that $e_0 = e_{\text{start}}$ and $\sigma_{k+1} \in \mathbb{S}(e_k)$ for all $0 \le k < N$.

Intuitively, an $\mathbb{S}$-interaction is a sequence of expressions where each one is the result of applying a step provided by $\mathbb{S}$ to the previous expression.

We also introduce notation for the set of valid expressions that are reachable from an expression.

Definition 3.4. The **completion** of an expression e is $C(e) = \{e' \mid e \preceq e' \land e' \text{ valid}\}$.

In general, the completion of an expression can be infinite, so completions are not computable in practice. Rather, we will use completions to describe the theoretical guarantees and requirements of Programming by Navigation synthesizers. Indeed, we can now precisely state what properties the set of steps returned by a Programming by Navigation synthesizer must satisfy.

Definition 3.5. A step set Σ **covers** an expression e if it satisfies the following three properties.

(1) **Validity**: $e \vdash \Sigma$.
(2) **Strong Completeness**: $\bigcup_{\sigma \in \Sigma} C(\sigma e) \supseteq C(e) \setminus \{e\}$.
(3) **Strong Soundness**: $C(\sigma e) \neq \emptyset$ for all $\sigma \in \Sigma$.

Validity ensures that all steps in Σ actually do something to e. **Strong Completeness** ensures that every valid program that was reachable from e is still reachable from at least one of the steps in Σ.[9] **Strong Soundness** ensures that all steps in Σ lead to at least one valid solution.

Finally, we can state the Programming by Navigation Synthesis Problem.

Problem Statement

Definition 3.6. A step provider $\mathbb{S}$ solves the **Programming by Navigation Synthesis Problem** if $\mathbb{S}(e_N)$ covers e_N for all $\mathbb{S}$-interactions $e_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_N} e_N$.

Figure 4 illustrates how a Programming by Navigation step provider can be used to navigate to a particular program. Intuitively, the step provider and step decider can be thought of as communicating coroutines. Starting with a blank program, e_{start}, the step provider provides a set of steps that covers the current working sketch. The step decider is free to choose among any of these

[8]For concision, we provide all proofs in Appendix B in the supplementary materials.
[9]The left-hand side in **Strong Completeness** is a (non-strict) subset of the right-hand side by **Validity**.

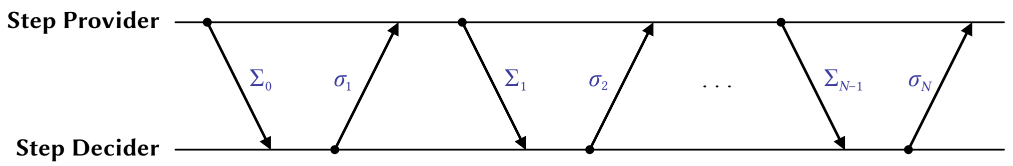

Fig. 4. **A step provider and step decider communicate to arrive at a particular valid solution.** The step provider is obligated to give a set Σ_i of steps that *covers* the working sketch at each stage (Definition 3.5). The step decider chooses $\sigma_{i+1} \in \Sigma_i$ (all of which are valid), which gets applied to working sketch. The resulting new sketch is fed back to step provider, and the process repeats until a valid program is constructed.

steps to be applied to the working sketch; none lead to a dead end. This updated sketch is then sent back to the step provider, and the process is repeated until reaching a particular valid program. Figure 2 gives an example interaction of the concrete instantiation we describe in Section 3.1.1.

This process is a form of specification refinement in that the set of programs satisfying the specification gets smaller at each round until the single particular program is left: $C(e_0) \supseteq C(e_1) \supseteq \cdots$. **Strong Soundness** ensures that these refined specifications never collapse into unsatisfiability.

3.3 Properties of Programming by Navigation

We now discuss the properties that all Programming by Navigation synthesizers $\mathbb{S}$ offer.

First, we note that **Strong Soundness** means that it is impossible for a step choice to flip a satisfiable specification into an unsatisfiable one. The following theorem captures the additional property that, if a specification is unsatisfiable, the Programming by Navigation process will immediately terminate.

Theorem 3.7 (Fail Fast). *If there are no valid expressions, then $\mathbb{S}(e_{start}) = \varnothing$.*

Furthermore, at each step of a Programming by Navigation interaction, either the current working sketch is a valid solution (and the interaction can finish) or the Programming by Navigation synthesizer will provide additional steps. This theorem is analogous to the traditional progress theorem of the λ-calculus [104].

Theorem 3.8 (Progress). *If there is at least one valid expression and $e_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_N} e_N$ is an $\mathbb{S}$-interaction, then either e_N valid or $\mathbb{S}(e_N) \neq \varnothing$ (or both).*

Finally, when we defined a navigation relation in Definition 3.1, we required all valid programs be reachable via some series of steps from the starting program (**Reachability**). What the following theorem establishes is that it is possible to construct any valid expression *using only the steps provided by a Programming by Navigation synthesizer.* This theorem is analogous to Omar et al. [78]'s constructability theorem for the Hazelnut structure editor calculus.

Theorem 3.9 (Constructability). *If e valid, there exists an $\mathbb{S}$-interaction $e_0 \xrightarrow{\sigma_1} \cdots \xrightarrow{\sigma_N} e$.*

4 Classical–Constructive Synthesis

We now describe our solution to the concrete instantiation of Programming by Navigation from Section 3.1.1. As we discuss in Section 2, we rely crucially on the notion of an inhabitation oracle, which, for each pair of a hole and a function, must determine if the hole can be extended by the function. (Later, when we model types in Section 5.1, this will correspond to checking whether

Algorithm 1 Top–Down Classical–Constructive Synthesis

Parameter: An inhabitation oracle $\mathcal{O}$ (Definition 4.1)
Input: An expression e
Output: A step set

```
1: procedure $\mathbb{S}(e)$
2:     for $(h, f) \in \mathcal{O}(e)$ do
3:         $?_{h_1}, \dots, ?_{h_{\mathrm{arity}(f)}} \leftarrow$ fresh$(e)$
4:         yield $(?_h \rightsquigarrow f(?_{h_1}, \dots, ?_{h_{\mathrm{arity}(f)}}))$
```

certain types are inhabited.) A function can only replace a hole if there exists a valid expression that is reachable from the resulting expression, as captured in the following definition.

Definition 4.1. An **expansion** is a hole name and function pair. An **inhabitation oracle** $\mathcal{O}$ maps expressions e to finite sets of expansions that include (h, f) if and only if

$$?_h \triangleleft e \qquad \text{and} \qquad C([?_h \mapsto f(?_{h_1}, \dots, ?_{h_{\mathrm{arity}(f)}})]e) \neq \varnothing \text{ for } ?_{h_1}, \dots, ?_{h_{\mathrm{arity}(f)}} \text{ fresh in } e.$$

An inhabitation oracle tells us whether a certain expansion is possible—a "yes" or "no" response, in the style of classical logic—but it does not construct a solution. However, we can use it as part of a step provider to solve the Programming by Navigation Synthesis Problem. This is precisely what we do in Algorithm 1. By construction, the steps from Algorithm 1 satisfy **Validity** and **Strong Soundness** when it is used as a Programming by Navigation synthesizer. It turns out that these steps are sufficiently broad to ensure **Strong Completeness** as well. Consequently:

Theorem 4.2 (Soundness). *Algorithm 1 solves the Programming by Navigation Synthesis Problem.*

5 Datalog-Backed Inhabitation Oracle

We now turn to the task of implementing an inhabitation oracle. To do so, we start in Section 5.1 by formalizing the particular notion of validity we used in Section 2; we note, however, that Algorithm 1 is a Programming by Navigation synthesizer regardless of the notion of validity chosen. We then show in Section 5.2 that we can define an inhabitation oracle for this notion of validity using Datalog that does not require constructing full solutions to synthesis subtasks.

Throughout this section, we use the notational shorthand $\overline{X} = X_1, \dots, X_N$ for some N.

5.1 Formalizing Validity

Figure 5 defines the syntax we used for validity in Section 2. We call this syntax $\mathsf{HB}_{\mathsf{core}}$, as it is the core syntax of Honeybee, our implementation of Programming by Navigation that we evaluate in Section 7. As in Section 2, types and atomic propositions are parameterized by values, and the set of functions for this instantiation is drawn from a set of base functions coupled with *output metadata*, a tuple of values notated with a superscript. Function signatures are annotated with formulas, which can be true ($\top$), equality relations, less-than relations, atomic propositions, or a conjunction of other formulas. For notational simplicity, all values are numeric, but the type system could be extended to handle other types such as strings (as we do in our actual implementation) or enumerations. Lastly, we assume a fixed arity map arity$(\cdot)$ for atomic propositions and types and, for auxiliary technical reasons later, a distinguished type $\tau^\star$ with arity 0.

Figure 6 defines what it means for an expression to be well-typed with the rule Well-Typed/Fun. In addition to having the types of arguments match the types of the corresponding function parameters (premises 1 and 4), all validity conditions of the functions present in an expression must be satisfied (premise 3). For tractability, we additionally require that all values be literals

$$\begin{array}{rl}
\textbf{Values } v \in \mathbb{Z} & \textbf{Base functions } f \\
\textbf{Functions } f^{\overline{v}} & \textbf{Type names } \tau \\
\multicolumn{2}{c}{\textbf{Atomic proposition names } \delta}
\end{array}$$

$$\begin{array}{rl}
\textbf{Types} & \tau(\overline{v}) \\
\textbf{Atomic propositions} & \delta(\overline{v}) \\
\textbf{Formula atoms } a & ::= \mathsf{param}_{i,j} \mid \mathsf{ret}_j \mid v \\
\textbf{Formulas } \varphi & ::= \top \mid a = a \mid a < a \\
& \mid \ \delta(\overline{a}) \mid \varphi \wedge \varphi \\
\textbf{Function libraries } \Gamma & ::= \{f_i \mapsto (\overline{\tau_i} \rightarrow_{\varphi_i} \tau_i)\}_{i=1}^{N}
\end{array}$$

Fig. 5. **HB$_{\text{core}}$, the core syntax of Honeybee.** Of note, functions $f^{\overline{v}}$ are coupled with output metadata (tuples of values) that constrain their type.

$\boxed{\Gamma, \Delta \vdash e : \tau(\overline{v})}$ e has type $\tau(\overline{v})$ with library Γ assuming atomic propositions Δ

Well-Typed/Fun

$$\frac{\begin{array}{c}\Gamma(f) = \tau_1, \ldots, \tau_N \rightarrow_{\varphi} \tau \\ \forall i, j.\, v_{ij} \in \mathsf{vals}(\Gamma) \cup \mathsf{vals}(\Delta) \cup \mathsf{vals}(\overline{v}) \\ \Delta \models \varphi[\overline{v_1}, \ldots, \overline{v_N}; \overline{v}] \\ \forall i.\, \Gamma, \Delta \vdash e_i : \tau_i(\overline{v_i})\end{array}}{\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})}$$

Fig. 6. **The typing relation for HB$_{\text{core}}$.** Function applications are well-typed if their arguments are well-typed and have metadata satisfying the validity condition. The notation $\mathsf{vals}(X)$ returns the set of values in X.

that appear in the library or synthesis problem (premise 2). As Well-Typed/Fun applies only to function applications, incomplete programs (those with holes) are not considered well-typed.

We now formalize the up-front specification from Section 2.

Definition 5.1. An **HB$_{\text{core}}$ problem** is a triple $(\Gamma, \Delta, \tau(\overline{v}))$ where Γ is a function library to use, Δ is a set of atomic propositions assumed to be true, and $\tau(\overline{v})$ is a goal type. For a fixed HB$_{\text{core}}$ problem $(\Gamma, \Delta, \tau(\overline{v}))$, we say e valid if $\Gamma, \Delta \vdash e : \tau(\overline{v})$.

We now need an oracle that, given an e as defined above, can determine if an expansion of it is well-typed according to Well-Typed/Fun in Figure 6. For example, if we have the expression $f^1(g^2(), ?_1, ?_2)$ and we are considering expanding $?_1$ with f, we need to determine if there exist expressions e_1, e_2, e_3, e_4, and a value v such that $f^1(g^2(), f^v(e_1, e_2, e_3), e_4)$ is well-typed. We essentially want to extend the Well-Typed/Fun rule so that the fourth premise allows for the possibility that e_i is a hole, in which case it would require that there exists some e_i' that can replace it that is well-typed instead. The key difficulty is that this change makes type-checking highly non-syntax-directed, essentially amounting to program search. Thus, we would like to be able to make this check without actually constructing such an expression e_i' to replace the hole with.

5.2 Compiling to Datalog

To construct such an oracle for an HB$_{\text{core}}$ problem (consisting of a library, set of atomic propositions, and goal type), we compile the problem to a Datalog program in which atomic propositions are the EDBs, types are the IDBs, and functions between the types get compiled to rules. We perform this compilation in such a way that a type is inhabited if and only if the corresponding Datalog fact is derivable. In fact, the key invariant (Lemma 5.4) that we maintain will be slightly stronger, asserting that a type is inhabited with an expression whose root is f if and only if the corresponding Datalog fact is derivable with a proof tree ending in the Datalog rule corresponding to f.

We first describe the notation we use to refer to Datalog (the technical content is standard).

Definition 5.2. A **Datalog program** is a triple $(\mathcal{V}, \mathcal{E}, \mathcal{R})$, where $\mathcal{V}$ is a value domain, $\mathcal{E}$ is a set of EDBs (ground facts), and $\mathcal{R}$ is a set of rules to derive IDBs (derived facts). For simplicity, we assume Datalog programs have EDBs corresponding to atomic propositions and IDBs corresponding to types from Section 5.1, as well as a distinguished fact $Q^\star$ used only for queries. The **Datalog provability relation** $\mathcal{V}, \mathcal{E}, \mathcal{R} \vdash_{\text{DL}} I(\overline{v})$ over Datalog programs $(\mathcal{V}, \mathcal{E}, \mathcal{R})$ with I an IDB and $v_1, \ldots, v_N \in \mathcal{V}$ holds if and only if $I(\overline{v})$ is derivable using the rules from $\mathcal{R}$ assuming the EDBs $\mathcal{E}$ are true and all values are drawn from $\mathcal{V}$. Lastly, a **Datalog engine DL** takes in a Datalog program $(\mathcal{V}, \mathcal{E}, \mathcal{R})$ and query rule R with head $Q^\star$ and returns all tuples of values $\overline{v}$ such that $\mathcal{V}, \mathcal{E}, \mathcal{R} \cup \{R\} \vdash_{\text{DL}} Q^\star(\overline{v})$.

We now define how to compile an HB_{core} problem to Datalog. We start with setting up rules that establish when types are inhabited (i.e., general versions of the $\textsc{Rule}_f$ rules from Section 2). We call these rules the *Datalog header* of a function library, as they will be included in all calls to the Datalog engine throughout the synthesis procedure.

Definition 5.3 (Datalog Compilation, Part 1/2). The **Datalog header** of a function library Γ is

$$\mathcal{H}[\![\Gamma]\!] = \left\{ \frac{\tau_1(\overline{x_1}) \quad \cdots \quad \tau_N(\overline{x_N}) \quad \varphi[\overline{x_1}, \ldots, \overline{x_N}; \overline{x}]}{\tau(\overline{x})} \;(\textsc{Rule}_f) \mid \Gamma(f) = \tau_1, \ldots, \tau_N \rightarrow_\varphi \tau \right\}.$$

This definition enables us to prove the following key invariant that links type inhabitation with Datalog fact derivability.

Lemma 5.4 (Key Invariant I). *The following are equivalent:*

(1) $vals(\Gamma) \cup vals(\Delta), \Delta, \mathcal{H}[\![\Gamma]\!] \vdash_{DL} \tau(\overline{v})$ with a derivation tree ending in $\textsc{Rule}_f$.

(2) There exist expressions $e_1, \ldots, e_N$ such that $\Gamma, \Delta \vdash f^{\overline{v}}(e_1, \ldots, e_N) : \tau(\overline{v})$.

Notably, in (1), Datalog can prove that a certain type is inhabited *without searching for the witness expressions in (2)*; this is precisely the benefit we get by requiring only a "yes" or "no" answer.

Now, given an expression like $f^1(g^2(), ?_1, ?_2)$ and goal $?_1$, we need a query rule that returns all v such that there exist expressions e_1 (with metadata v) and e_2 that can replace $?_1$ and $?_2$ to make the expression well-typed (i.e., a general version of the $\textsc{Query}$ rule from Section 2). This rule should check (i) the types of all hole siblings of $?_1$ (including the type of $?_1$ itself) are inhabited with metadata values that satisfy the validity condition of f on its arguments, and (ii) the function siblings (e.g., $g^2()$) recursively satisfy this property. The following definition achieves this goal.

Definition 5.5 (Datalog Compilation, Part 2/2). The set of **Datalog queries** of an expression e given a function library Γ and expected type τ is

$$\mathcal{Q}_{\Gamma,\tau}[\![e]\!] = \left\{ \frac{\mathcal{P}^{\cdot}_{\Gamma,\tau}[\![e]\!]}{Q^\star(\overline{x_p})} \;(\textsc{Query}_{h,k}) \mid ?_h \triangleleft e \text{ at path } p \text{ and } \tau'(\overline{x_p}) \text{ is the } k\text{th premise} \right\}$$

where the premises are defined by

$$\mathcal{P}^{p}_{\Gamma,\tau}[\![?_h]\!] = \tau(\overline{x_p}) \quad \text{and} \quad \mathcal{P}^{p}_{\Gamma,\tau}[\![f^{\overline{v}}(e_1, \ldots, e_N)]\!] = (\overline{x_p} = \overline{v}) \wedge \varphi[\overline{x_{p,1}}, \ldots, \overline{x_{p,N}}; \overline{x_p}] \wedge \bigwedge_{i=1}^{N} \mathcal{P}^{p,i}_{\Gamma,\tau_i}[\![e_i]\!]$$

for $\Gamma(f) = \tau_1, \ldots, \tau_N \rightarrow_\varphi \tau$.

Intuitively, the $\textsc{Query}_{h,k}$ rule for an expression e will hold on $\overline{v}$ if and only if there exists an expression e' with metadata value $\overline{v}$ to replace $?_h$ such that all validity conditions either hold or can hold with further expansions. We use Lemma 5.4 to prove the following additional key invariant that captures this idea more formally.

Lemma 5.6 (Key Invariant II). *Suppose $?_h \triangleleft e$. Then the following are equivalent:*

(1) $\textsc{Query}_{h,k} \in \mathcal{Q}_{\Gamma,\tau}[\![e]\!]$ and $vals(\Gamma) \cup vals(\Delta), \Delta, \mathcal{H}[\![\Gamma]\!] \cup \{\textsc{Query}_{h,k}\} \vdash_{DL} Q^\star(\overline{v})$ with a derivation tree whose kth subtree ends in $\textsc{Rule}_g$.

(2) There exist expressions $e_1, \ldots, e_N$ such that $C([?_h \mapsto g^{\overline{v}}(e_1, \ldots, e_N)]e) \neq \emptyset$.

As before, in (1), Datalog can prove that a hole can be replaced with a well-typed function application *without searching for the witness expressions in (2).* However, there is one final challenge: To make these queries into an oracle, we need to be able to check for specific choices of g.

Algorithm 2 Datalog-Backed Inhabitation Oracle

```
Parameter: A Datalog engine DL (Definition 5.2)
Parameter: An HB_core problem $(\Gamma, \Delta, \tau(\overline{v_{\text{goal}}}))$
Input: An expression $e$
Output: A finite set of expansions
1: procedure $\mathcal{O}(e)$
2:     $\Gamma' \leftarrow \Gamma \cup \{\text{goal} \mapsto (\tau \rightarrow_{(\text{param}_{1,1}=v_{\text{goal},1} \wedge \cdots \wedge \text{param}_{1,N}=v_{\text{goal},N})} \tau^\star)\}$
3:     for $\textsc{Query}_{h,k} \in \mathcal{Q}_{\Gamma',\tau^\star}[\![\text{goal}^{()}(e)]\!]$ do
4:         for $\textsc{Rule}_f \in \mathcal{H}[\![\Gamma]\!]$ do
5:             if $\textsc{Rule}_f$ not $k$-compatible with $\textsc{Query}_{h,k}$ then continue
6:             for $\overline{v} \in \mathbf{DL}((\text{vals}(\Gamma') \cup \text{vals}(\Delta), \Delta, \mathcal{H}[\![\Gamma]\!]), \textsc{Rule}_f/k/\textsc{Query}_{h,k})$ do
7:                 yield $(h, f^{\overline{v}})$
```

We can force g to be a specific function by using the notion of a *logical cut*, as defined below. Using a cut, we can control exactly what the final derivation rule of a type is going to be, and thus, by Lemma 5.4, what the function at the head is going to be.

Definition 5.7 (Cut). Suppose $1 \le k \le N$ and consider the Datalog rules

$$\frac{P_1(\overline{x_1}) \quad \cdots \quad P_M(\overline{x_M}) \qquad \varphi_P[\overline{x_1}, \ldots, \overline{x_M}; \overline{x}]}{P(\overline{x})}\ (R_1) \quad \text{and} \quad \frac{Q_1(\overline{y_1}) \quad \cdots \quad Q_N(\overline{y_N}) \qquad \varphi_Q[\overline{y_1}, \ldots, \overline{y_N}; \overline{y}]}{Q(\overline{y})}\ (R_2),$$

where the IDB premises of R_1 are $P_1, \ldots, P_M$ and the IDB premises of R_2 are $Q_1, \ldots Q_N$. If $P = Q_k$, we say R_1 is k-**compatible** with R_2, and the k-**cut** of R_2 with R_1 is the rule

$$\frac{Q_1(\overline{y_1}) \quad \cdots \quad Q_{k-1}(\overline{y_{k-1}}) \quad P_1(\overline{x_1}) \quad \cdots \quad P_M(\overline{x_M}) \quad Q_{k+1}(\overline{y_{k+1}}) \quad \cdots \quad Q_N(\overline{y_N}) \qquad \varphi_P[\overline{x_1}, \ldots, \overline{x_M}; \overline{x}] \quad \varphi_Q[\overline{y_1}, \ldots, \overline{y_N}; \overline{y}] \quad \overline{y_k} = \overline{x}}{Q(\overline{y})}\ (R_1/k/R_2).$$

To establish correctness of this approach, we prove the following analogue of cut elimination [36].

Lemma 5.8. *Let R_1 and R_2 be as above. Then $\overline{v}$ satisfies the query rule R_2 with a derivation tree whose kth subtree ends in R_1 if and only if $\overline{v}$ satisfies the query rule $R_1/k/R_2$.*

We can now assemble these three parts—Datalog headers, Datalog queries, and cut—in Algorithm 2. First, we wrap the input expression e in a function goal whose validity condition enforces that the output type has the right metadata (and whose output type uses the auxiliary type $\tau^\star$ we mentioned earlier). Then, we loop through each of the queries of e (one per hole). If we did not need to determine precisely which function to use for a hole expansion, we could simply run the Datalog engine on these queries. But, since we must determine which functions, specifically, are valid expansions, we loop through all rules in the header (one per function in the library), and take only those that are compatible with the query at the position of the hole. This amounts to taking the functions whose return type is the required parameter type. Finally, we run the Datalog engine on the cut of the query with each matching header rule to determine for which values (if any) it is possible to extend the expression at the given hole using the given function.

By using Lemma 5.6, we can prove the following theorem that finishes our solution to the Programming by Navigation Synthesis Problem.

Theorem 5.9. *Algorithm 2 is an inhabitation oracle.*

6 Implementation

We implemented Programming by Navigation for $\mathrm{HB}_{\mathrm{core}}$ using Algorithms 1 and 2 in a system called Honeybee in approximately 2,300 lines of Rust code. We used the `egglog` library [109] without any equality saturation features for our Datalog engine, although our algorithm (and implementation) is agnostic to the underlying Datalog engine.

One benefit of using a Datalog-backed oracle is that some standard program synthesis optimizations fall out of existing Datalog engines essentially for free. For example, as we will see in Section 7.3, repeated calls to a Datalog engine can be optimized by simply not discarding derived facts from previous runs. This amounts to the synthesis optimization of subtask memoization, in which synthesis results from recursive calls to the synthesizer are retained. Although `egglog` does not currently support it, another applicable off-the-shelf Datalog optimization is the magic sets optimization [9], which would correspond to the synthesis optimization of top-down enumeration.

Remark 6.1. We implement one additional optimization. Honeybee only needs to check a subset of query premises at each step. Specifically, it can (non-recursively) check only the premises in Definition 5.5 that arise from the target's parent expression and the parent's immediate children. This is because Datalog has already checked the other premises in prior rounds of Algorithm 1.

7 Evaluation

To evaluate Honeybee as an instantiation of Programming by Navigation, we were most interested in the following two research questions:

RQ1 How well does Honeybee solve the Programming by Navigation Synthesis Problem?
RQ2 How does Honeybee scale with problem size?

Although they are not our main focus, we were also curious to explore the following two additional research questions for further empirical context:

RQ3 What effect does using off-the-shelf Datalog memoization have on performance?
RQ4 How does Honeybee perform on the traditional Any task?

As the Programming by Navigation Synthesis Problem and our instantiation of the framework for the top-down setting are both novel, there is no existing benchmark that we can evaluate our approach on. We therefore constructed three suites of types and functions for this evaluation:

(1) **Fin**, which consists of 13 problems with finitely many solutions. Two of these are biological analysis tasks, one of which is a slight extension of the Section 2 example, and one of which models analyzing data from a pooled CRISPR screen [16, 91, 101]. The remainder are generic, in the style of the example from Section 2 but with no particular domain encoded.
(2) **Inf**, which consists of 8 problems with infinitely many solutions. Three of these are geospatial analysis tasks, one of which is to compuate a normalized difference vegetation index (NDVI) [89] using imagery from the Landsat-9 satellite [69], and two of which perform raster geoprocessing operations, including warping (reprojection), resampling, and clipping with vector masks. The remainder are generic.
(3) **Scal**, which consists of programmatically-generated problems that differ only in the depth and breadth of their search space.

Each entry in these suites consists of an $\mathrm{HB}_{\mathrm{core}}$ problem (a library, set of atomic propositions, and goal type) as well as a fixed set of 10 particular programs that are solutions to the problem (or all solutions if there are less than 10). These solutions are the *particular* programs that a synthesizer must return to solve the Programming by Navigation Synthesis Problem (but not the Any task). We evaluate Honeybee by taking the median of 10 measurements of execution time per $\mathrm{HB}_{\mathrm{core}}$ problem—one for each particular program. Each step in the interactive synthesis process is chosen

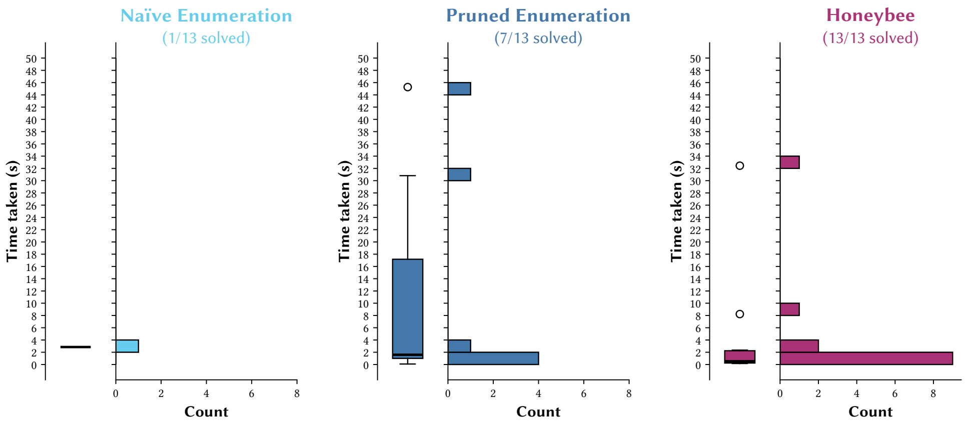

Fig. 7. **Completion rates for benchmarks with *finitely* many solutions (the Fin suite).** Honeybee solves the most (13/13), followed by Pruned Enumeration (7/13), and lastly Naïve Enumeration (1/13).

programmatically based on the particular program at hand. Each time measurement is the median of 5 identical runs of the synthesizer. We emphasize that these Honeybee measurements are total synthesis time, not time per round.

We constructed these 10 particular programs for each entry once, ahead-of-time, in one of two ways. For **Fin**, **Scal**, and all but one of the generic benchmarks from **Inf**, we ran Honeybee on each synthesis problem and selected from among the provided steps at each round uniformly at random, and recorded the first 10 unique programs found. For one of the generic benchmarks and all of the geospatial benchmarks in **Inf**, this random sampling procedure did not terminate, so we manually navigated to 10 unique programs.

To contextualize our results, we also implemented two baseline algorithms, Naïve Enumeration and Pruned Enumeration. To solve the Programming by Navigation Synthesis Problem, these baseline algorithms enumerate *all* solutions to the synthesis task. Therefore, they are only applicable to the **Fin** and **Scal** suites, as the **Inf** problems have infinitely many solutions. Both Naïve Enumeration and Pruned Enumeration explore the search space in a breadth-first fashion and check candidate solution validity post hoc. The difference between the two is that Pruned Enumeration additionally attempts to prune the synthesis space during exploration by not considering paths that violate an existing validity condition in the candidate solution. As these baseline algorithms do not take into consideration a particular program, we simply take the median execution time of 5 identical runs of the synthesizer per entry in the benchmark.

We ran the evaluation on a system running Ubuntu 20.04.6 LTS with an Intel Xeon E5-2650 v2 CPU at 2.60 GHz and 64 GB of RAM. All synthesis tasks had a time cutoff of 2 minutes. All system implementations are single-threaded, so the evaluation ran entirely on one CPU core.

7.1 RQ1 Results: Honeybee Solves Benchmarks Impossible or Too Large for Baselines

Figure 7 shows the results of running the algorithms on the **Fin** suite. Naïve Enumeration solves 1/13 of the benchmarks, Pruned Enumeration solves 7/13, and Honeybee solves all 13/13.

Figure 8 shows the results of running Honeybee on the **Inf** suite. As mentioned above, neither Naïve Enumeration nor Pruned Enumeration can solve these benchmarks because it is impossible to enumerate all solutions. Honeybee solves 8/8 of the benchmarks with infinitely many solutions in a similar amount of time to the entries from the **Fin** suite.

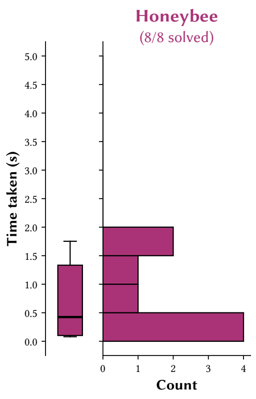

Fig. 8. **Completion rates for benchmarks with *infinitely* many solutions (the Inf suite).** Due to the requirement of Strong Completeness, techniques based on exhaustive enumeration cannot solve these benchmarks. Honeybee solves 8/8 benchmarks.

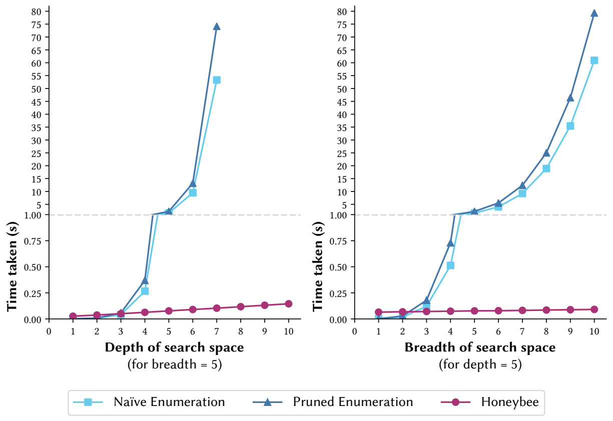

Fig. 9. **Time to solve the scalability benchmarks (the Scal suite).** The chart on the left shows execution time as we hold the breadth of the search space constant at 5 and vary depth from 1–10. The chart on the right shows execution time as we hold the the depth of the search space constant at 5 and vary breadth from 1–10. On small programs, the Naïve Enumeration and Pruned Enumeration perform better than Honeybee due to the overhead of Programming by Navigation, but—unlike Honeybee—they do not scale to larger problems.

7.2 RQ2 Results: Honeybee Scales to Large Problems While Naïve Enumeration and Pruned Enumeration Solve Smaller Problems Faster

Figure 9 shows the results of running the algorithms on the **Scal** suite. We varied the depth of the search space from 1 to 10 while keeping the breadth fixed at 5. Similarly, we varied the breadth of the search space from 1 to 10 while keeping the depth fixed at 5.

We observe that Honeybee scales linearly with the depth of the search space. This is expected of these benchmarks: a depth of N requires N calls to the oracle. Honeybee's performance is not substantially affected by search space breadth at these scales; depth has a much larger impact.

Neither Naïve Enumeration nor Pruned Enumeration scale well with either the depth or breadth of the search space. However, for small tasks, the overhead of Programming by Navigation dominates the runtime. Additionally, for these simple scalability tasks, pruning overhead causes Pruned Enumeration to be slightly slower than Naïve Enumeration.

7.3 RQ3 Results: Honeybee Benefits From Off-The-Shelf Memoization

To see the impact of an off-the-shelf Datalog optimization on Honeybee's performance, we created an ablated version of Honeybee that does not use the off-the-shelf memoization optimization we describe in Section 6. Figure 10 shows the results of running these two synthesizers on all benchmarks from all benchmark suites (**Fin**, **Inf**, and **Scal**). We use all benchmarks from all three benchmark suites, filtered only based on which benchmarks the slower ablated Honeybee can solve. On the 37/40 benchmarks that the ablated version completed, the full version was always faster, with a median speedup of approximately 6.9×.

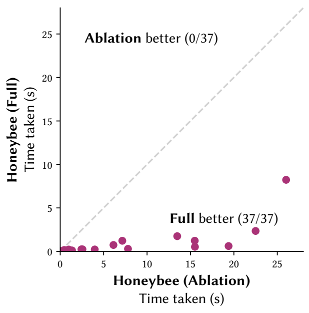

Fig. 10. **Inhabitation oracle performance with and without off-the-shelf Datalog memoization.** The y-axis shows time to solve benchmarks with memoization (full Honeybee), and the x-axis shows time to solve benchmarks without memoization (ablated Honeybee). Full Honeybee solves all benchmarks faster.

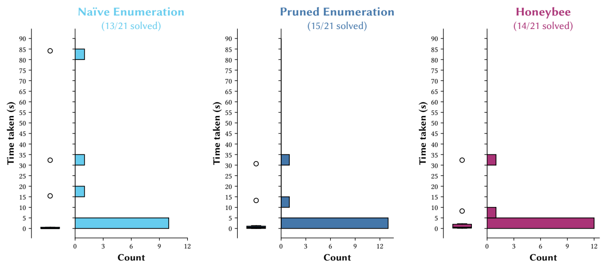

Fig. 11. **Completion rates for benchmarks from the Fin and Inf suites on the Any task, for which Programming by Navigation was not designed.** Pruned Enumeration solves the most (15/21), followed by Honeybee (14/21), then Naïve Enumeration (13/21). As a consequence of satisfying **Strong Completeness**, Honeybee does not perform as well on the Any task as it does on the Particular task, as we discuss in Section 7.4.

7.4 RQ4 Results: On the Any Task, Honeybee Performs Comparably to Baselines

Programming by Navigation was not designed to solve the Any task. Nevertheless, we were interested in what overhead the Programming by Navigation approach would entail when used to solve the Any task rather than the Programming by Navigation Synthesis Problem. To solve the Any task with Naïve Enumeration and Pruned Enumeration, we ended enumeration after the first solution they found. To solve the Any task with Honeybee, we automatically selected the first of the available steps provided in each synthesis round. We ran all three algorithms on both the **Fin** and **Inf** suites, as the baselines need not enumerate all solutions to solve the Any task.

Figure 11 shows that Naïve Enumeration solves 13/21 of the benchmarks, Pruned Enumeration solves 15/21, and Honeybee solves 14/21. As expected, Naïve Enumeration and Pruned Enumeration can solve *more* benchmarks on the Any task than when solving the Programming by Navigation Synthesis Problem, as, even on the benchmarks where they cannot return *all* solutions, they may be able to find *some* solution. Additionally, due to always selecting the first step, Honeybee solves *fewer* benchmarks on the Any task. We can think of the select-first-step strategy as essentially performing depth-first search in the space of steps. While it is always valid to select the first step (a valid program is still available along the path, even if we end synthesis at the time cutoff), doing so may extend the interactive process compared to selecting other steps. This is a necessary consequence of the requirement to be able to reach all solutions (**Strong Completeness**). In comparison, for performance reasons, we specifically designed our enumerative approaches to be breadth-first so that they would always arrive at a solution of minimal size first.

7.5 Threats to Validity

As both our synthesis problem and instantiation are novel, there is no existing benchmark suite. As a result, we can artificially inflate the number of benchmarks that Honeybee solves compared to the baseline approaches. For example, in the extreme, we could exclude any benchmark from **Fin** that the baselines could solve. To partially mitigate this potential bias, we randomly sampled the 10 particular programs associated with each benchmark for all but four benchmarks. For these four benchmarks (all in the **Inf** suite), we could not randomly sample programs, so we manually selected the particular programs for these benchmarks. Since we selected the programs, it is possible that we specifically selected programs that HONEYBEE performs particularly well on. We mitigated this bias by not measuring HONEYBEE performance on any of the selected particular programs until the programs had already been fixed. Additionally, HONEYBEE is the only tool that solves the Programming by Navigation Synthesis Problem when there are infinitely many solutions, so these manual choices cannot disadvantage our baselines.

8 Limitations

Programming by Navigation requires strong guarantees of the synthesis process—**STRONG COMPLETENESS** and **STRONG SOUNDNESS**—which may not always be possible to adhere to in designing a synthesis algorithm. For example, when working with fully dependent types such as in Rocq [88], Agda [76], or Lean [73], type inhabitation is undecidable, so maintaining both **STRONG COMPLETENESS** and **STRONG SOUNDNESS** is provably impossible. Indeed, Löb [60] showed that type inhabitation in even just System F is undecidable. Therefore, there will always be a place for synthesizers that do not offer the guarantees of Programming by Navigation.

Additionally, Section 7.2 shows that Programming by Navigation has overhead compared to enumeration. For smaller problems, enumeration remains a strong strategy for program synthesis.

9 Discussion

In the sections above, we have developed a formalism, problem statement, and algorithmic solution for a new interactive program synthesis approach. We do not have evidence that this approach is usable by real-world users, and this paper makes no claims about its usability. In fact, the step decider in our approach (Figure 4) does not need to be a human user at all! Having said this, we are excited about the possibility of building a usable interface for Programming by Navigation and briefly discuss some related topics below.

9.1 Backtracking in Programming by Navigation

Programming by Navigation guarantees **STRONG SOUNDNESS**, which ensures that the current specification is always satisfiable at every stage of the iterative refinement process. However, a user may still wish to opt-in to backtracking for at least two reasons:

(1) **A user might realize a specific previously-taken step was a mistake.** Let's say the user selected $\sigma_1, \sigma_2, \sigma_3, \sigma_4$. If they realize σ_3 was a mistake, they could simply revert to σ_2 (saving their work up to that point) and continue Programming by Navigation as usual.

(2) **A user might realize that none of the currently-suggested steps are desirable.** Let's say the user has selected $\sigma_1, \sigma_2, \sigma_3, \sigma_4$ and the provided next steps are $?_1 \mapsto f(?_3)$ and $?_2 \mapsto g(?_4, ?_5)$, where these steps are drawn from our top-down instantiation of Programming by Navigation. If the user is unhappy with these options, the synthesizer could remove f and g from the library and check if the queries (Definition 5.5) for the holes in the working expression are derivable. (This requires only a single query to the Datalog engine per hole in the working expression.) With the working expression resulting from σ_4, this check is guaranteed to fail, as f and g are no longer in the library. The synthesizer can then revert to σ_3 and try this check again; if it succeeds, then the synthesizer has successfully identified the user-selected step (σ_4) that led them to the undesirable suggestions. Otherwise, the synthesizer can keep backtracking—even all the way to the starting expression. If the goal fact is still not derivable at the starting expression, then the problem has no solution using the new component library, which the tool can report to the user.

9.2 Programming by Navigation in Context

Finally, we briefly describe the constraints of the real-world setting that inspired Programming by Navigation and how it fits those constraints.

Programming by Navigation arose out of a close collaboration with experimental biologists who have immense domain expertise but limited programming experience. Taking the working example from Section 2, many of the biologists deeply understand the idea of batch-level differences between samples, but do not necessarily know that there is a computational way to correct for it or how to write a program to do so.

However, we observed that the biologists can easily and quickly provide details of their experiment. For example, for a pooled CRISPR screen [16], they might list the following:

- Transfect cells on day 1 using transfection sequences in file `transfect.fasta`.
- Sequence cells on day 3, results stored in file `day3.fastq`.
- Sequence cells on day 10, results stored in file `day10.fastq`.

Their goal would then be to calculate what is called a "phenotype score" between days 3 and 10. Compared to more complicated experiments, a CRISPR screen is relatively easy to analyze. However, just one step of the analysis pipeline alone—computing a phenotype score from processed data—can be done with at least 9 different algorithms [17]!

The above sequence of facts translates to atomic propositions in Programming by Navigation as

$$\Delta = \{\,\texttt{Transfect(1, “transfect.fasta”)}, \texttt{Sequence(3, “day3.fastq”)}, \texttt{Sequence(10, “day10.fastq”)}\,\}.$$

The goal of computing a phenotype score between days 3 and 10 corresponds to a goal type of $\texttt{PhenotypeScore}\langle 3, 10\rangle$. Programming by Navigation takes this specification as input, and provides the set of next steps that are possible for this specification. At a high level, here is what the first round of interaction would look like:

You want to calculate a **`PhenotypeScore`**. *Here are the options to do so:*

1) Redundant siRNA activity [52]
2) MAGeCK robust ranking algorithm [58]
3) HiTSelect [26]
4) MAGeCK maximum likelihood estimation [57]
5) BAGEL [40]
6) CRISPhieRmix [25]
7) CERES [71]
8) JACKS [5]
9) *t*-test

Which option would you like to choose?

In the above interaction, the synthesizer could also link to Bodapati et al. [17]'s comparison of these methods and the experimental biologist could use their expert judgment to decide which works best for their particular scenario. (The link to this comparison or other supporting information could easily be stored as an annotation on the `PhenotypeScore` type in the library.)

The biologist does not need to know how to write the program in order to use Programming by Navigation; they just need to decide between this curated list of options (which, by **Strong Soundness** and **Strong Completeness** is exactly the list of valid next steps). Moreover, they do not need to know these choices up front; they are provided with exactly the set of choices they need to make, and exactly the set of options available for each choice. Thus, Programming by Navigation here does not assume that the user knows (i) how to write the program nor (ii) the set of components involved in it. Instead, it assumes that the user can make an informed decision between different choices in the program when presented with exactly the list of valid next steps.

10 Related Work

10.1 Iterative Specification Refinement

Programming by Navigation is a type of interactive synthesis, in which an initial under-specification is iteratively refined. Programming by Navigation offers two guarantees: **Strong Completeness** and **Strong Soundness**. As a reminder, **Strong Completeness** ensures that all valid programs are constructible. **Strong Soundness** ensures that, at each round of interaction, all provided steps (which can be thought of as specification refinements) lead to a valid program; in other words, that a valid specification can never be refined to an unsatisfiable one.

Prior work offers different guarantees. For example, some approaches assume that the specification refinement is always done correctly (for some notion of correctness), which violates **Strong Soundness**. This includes minimally adequate teachers in Angluin [6]'s foundational L^* algorithm, Le et al. [55]'s ϕ^*-driven interactive program synthesis process (Problem 1, Axioms A and B), and Peleg et al. [83]'s iterative behavior requirement (Definitions 5 and 6). If it is not possible to maintain the guarantees of Programming by Navigation in a particular domain, such assumptions are very useful for maintaining theoretical properties such as soundness.

More generally, many interactive program synthesizers take in a specification, return a program, and enable arbitrary modifications to the specification before rerunning synthesis [44, 82, 84, 107, 110]. Peleg et al. [83] put this approach on a firm theoretical basis using abstract domains. Such approaches are highly flexible but, as above, do not guarantee **Strong Soundness**.

Another class of synthesizers enables pruning of the search space of the synthesizer as it runs, such as Zhang et al. [106]'s search tree interface and Blinn et al. [15]'s Hazel Live Assistant. Like Programming by Navigation, these approaches can guide the synthesizer to a particular solution, but they do not offer **Strong Soundness**.

A final class of interactive program synthesizers support specification refinement in the form of assigning output values to disambiguating examples. Jha et al. [46]'s oracle-guided inductive synthesis, Mayer et al. [70]'s conversational clarification interface, and Angluin [6]'s L^* algorithm all fall into this category. Additionally, a variety of prior work has explored optimal disambiguating questions to propose [21, 47, 80, 100]. As output values may be arbitrarily assigned to input values, inconsistent responses can yield an unsatisfiable specification, violating **Strong Soundness**.

Mayer et al. [70] proposes an interface that comes close to **Strong Soundness**; it lets users choose between a small number of alternatives for subexpressions of a synthesized term. However, it also allows arbitrary modifications, violating **Strong Soundness**. Further, not all expressions are guaranteed to be accessible. Consequently, it does not offer **Strong Completeness**.

10.2 Datalog for Programming Languages

Many prior works have used hand-written Datalog to express program analyses such as point-to analysis declaratively [19, 33, 90, 98, 103]. In contrast, we use programatically-generated Datalog as an oracle for our synthesizer. Another line of work aims to synthesize Datalog programs [4, 86, 93, 102]; we do the converse, synthesizing non-Datalog programs using Datalog under the hood. Lastly, a variety of Datalog extensions have brought additional expressive power to Datalog, often for the purposes of program analyses [7, 8, 12, 13, 66, 109]. An exciting future direction would be to leverage these techniques for efficient oracles for more expressive notions of validity.

10.3 Program Synthesis Techniques

Our particular instantiation of Programming by Navigation is for component-based synthesis, which builds on a long and rich history of prior work [1, 2, 11, 14, 18, 22, 30–32, 34, 37, 39, 42, 45, 46, 48, 53, 63, 67, 68, 75, 92, 94, 97]. Although we do not explore the idea here, it would be exciting future work to explore applying Programming by Navigation to recursive program synthesis (in which a recursive program is synthesized, often from scratch), which also has a long and rich history [3, 28, 29, 43, 49, 50, 56, 62, 72, 74, 79, 85, 105].

Programming by Navigation features an iterative loop between two interacting components: a step provider and a step decider. CEGIS [95, 96] and its generalization OGIS [46] also feature an iterative loop between two components: a program candidate proposer and a counterexample generator. The CEGIS/OGIS architecture differs in a number of ways from Programming by Navigation. The program candidate proposer synthesizes a single complete program that is consistent with a set of examples; in contrast, our step provider produces a set that includes all and only the valid next steps. The counterexample generator must prove that a program satisfies a specification or return a counterexample; in contrast, our step decider must simply choose among the set of valid next steps. Overall, CEGIS/OGIS follows a generate-and-check approach, whereas Programming by Navigation synthesizers satisfy **Strong Completeness** and **Strong Soundness**.

Lastly, Programming by Example synthesizers base their specification on input-output examples [3, 11, 32, 34, 37, 42, 43, 56, 62, 72, 74, 75, 79, 92, 94, 97, 105]. For domains with complex operations and data that is hundreds of gigabytes large—such as bioinformatics and geospatial analyses—it is not clear what an input-output example would constitute. However, it would be exciting to explore Programming by Navigation in domains where input-output examples could apply. It may be possible to draw on prior Programming By Example synthesizers' version space algebras (VSAs) [37, 55, 70] to define an oracle that uses a VSA-based internal representation rather than a Datalog-based one for problems with finitely many solutions.

10.4 Other Relevant Techniques

Structure editors [24, 35, 41, 51, 78, 87, 99] and other AST construction tools [10] prevent program edits that are syntactically invalid. Omar et al. [78]'s Hazelnut structure editor calculus goes a step further, modeling fine-grained cursor movements and ensuring that all edits result in well-typed programs. In a similar spirit, Programming by Navigation prevents steps that are *semantically* invalid with respect to some notion of validity. An interesting direction of future work could be to combine Programming by Navigation with Omar et al. [78]'s cursor traversal guarantees.

Additionally, the goals and steps displayed by Programming by Navigation (as shown in Figure 2, for example) have similarities to the proof state displayed by proof assistants such as Rocq [88] and Lean [73]. A key difference is that these proof assistants do not (and cannot in general, due to the undecidability) display a list of steps—the equivalent of which would be *tactics*—that would be guaranteed to advance the proof state toward the proof goal.

11 Conclusion

While existing techniques for refining underspecifications will remain the best fit for situations in which type inhabitation is undecidable, no prior technique offers **Strong Completeness** and **Strong Soundness**. This work introduces the Programming by Navigation Synthesis Problem, a new synthesis problem formulation designed to structure the process of iteratively refining a specification in order to find a particular solution. In contrast to prior specification-refinement synthesis techniques, we formally prove that synthesizers that solve the Programming by Navigation synthesis problem show *all* valid steps and *only* valid steps. By taking advantage of the fact that the synthesizer only needs to know *whether or not* a valid solution exists along a path, we develop a classical–constructive synthesis algorithm for solving the Programming by Navigation Synthesis Problem. We also propose and implement an inhabitation oracle for identifying valid paths without preemptively constructing programs. Together, these contributions represent the first specification refinement synthesizer that offers both **Strong Completeness** and **Strong Soundness**.

Data Availability Statement

Honeybee is open source and freely available at https://github.com/justinlubin/honeybee. We also provide an archived snapshot of the Honeybee codebase and evaluation setup (including the benchmark suite and all particular programs for each entry) for reproduction as a Docker image hosted on Zenodo [64, 65].

Acknowledgments

We are grateful for the feedback from our anonymous PLDI reviewers and shepherd. We are also deeply indebted to James Nuñez, Mercedes Paredes, Jennifer Listgarten, and all members of their labs for their generous and patient mentorship in biology. We would also like to thank Max Willsey for the insightful conversations and the developers of the `egglog` Rust crate for making a wonderful library. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. 175281 and is supported in part by NSF grants FW-HTF 2129008 and CA-HDR 2033558, as well as by gifts from the Chan Zuckerberg Biohub, Google, G-Research, Adobe, and Microsoft. Chasins is a Chan Zuckerberg Biohub Investigator.

References

[1] Maaz Bin Safeer Ahmad and Alvin Cheung. 2018. Automatically Leveraging MapReduce Frameworks for Data-Intensive Applications. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. doi:10.1145/3183713.3196891

[2] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. 2019. Automatically Translating Image Processing Libraries to Halide. In *ACM Transactions on Graphics (TOG)*. doi:10.1145/3355089.3356549

[3] Aws Albarghouthi, Sumit Gulwani, and Zachary Kincaid. 2013. Recursive Program Synthesis. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*. doi:10.1007/978-3-642-39799-8_67

[4] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *Principles and Practice of Constraint Programming*. doi:10.1007/978-3-319-66158-2_44

[5] Felicity Allen, Fiona Behan, Anton Khodak, Francesco Iorio, Kosuke Yusa, Mathew Garnett, and Leopold Parts. 2019. JACKS: Joint Analysis of CRISPR/Cas9 Knockout Screens. In *Genome Research*. doi:10.1101/gr.238923.118

[6] Dana Angluin. 1987. Learning Regular Sets from Queries and Counterexamples. In *Information and Computation*. doi:10.1016/0890-5401(87)90052-6

[7] Michael Arntzenius and Neel Krishnaswami. 2019. Seminaïve Evaluation for a Higher-Order Functional Language. In *Proceedings of the ACM on Programming Languages, Issue POPL*. doi:10.1145/3371090

[8] Michael Arntzenius and Neelakantan R. Krishnaswami. 2016. Datafun: A Functional Datalog. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming (ICFP)*. doi:10.1145/2951913.2951948

[9] Francois Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. 1986. Magic Sets and Other Strange Ways to Implement Logic Programs (Extended Abstract). In *Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS)*. doi:10.1145/6012.15399

[10] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 2021. Synthesizing Queries via Interactive Sketching. arXiv:1912.12659 [cs]

[11] Rohan Bavishi, Caroline Lemieux, Roy Fox, Koushik Sen, and Ion Stoica. 2019. AutoPandas: Neural-Backed Generators for Program Synthesis. In *Proceedings of the ACM on Programming Languages, Issue OOPSLA*. doi:10.1145/3360594

[12] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2020. Formulog: Datalog for SMT-based Static Analysis. In *Proceedings of the ACM on Programming Languages, Issue OOPSLA*. doi:10.1145/3428209

[13] Aaron Bembenek, Michael Greenberg, and Stephen Chong. 2024. Making Formulog Fast: An Argument for Unconventional Datalog Evaluation. In *Proceedings of the ACM on Programming Languages, Issue OOPSLA2*. doi:10.1145/3689754

[14] Sahil Bhatia, Sumer Kohli, Sanjit A. Seshia, and Alvin Cheung. 2023. Building Code Transpilers for Domain-Specific Languages Using Program Synthesis. In *37th European Conference on Object-Oriented Programming (ECOOP)*. doi:10.4230/LIPIcs.ECOOP.2023.38

[15] Andrew Blinn, David Moon, Eric Griffis, and Cyrus Omar. 2022. An Integrative Human-Centered Architecture for Interactive Programming Assistants. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. doi:10.1109/VL/HCC53370.2022.9833110

[16] Christoph Bock, Paul Datlinger, Florence Chardon, Matthew A. Coelho, Matthew B. Dong, Keith A. Lawson, Tian Lu, Laetitia Maroc, Thomas M. Norman, Bicna Song, Geoff Stanley, Sidi Chen, Mathew Garnett, Wei Li, Jason Moffat,

Lei S. Qi, Rebecca S. Shapiro, Jay Shendure, Jonathan S. Weissman, and Xiaowei Zhuang. 2022. High-Content CRISPR Screening. In *Nature Reviews Methods Primers*. doi:10.1038/s43586-021-00093-4

[17] Sunil Bodapati, Timothy P. Daley, Xueqiu Lin, James Zou, and Lei S. Qi. 2020. A Benchmark of Algorithms for the Analysis of Pooled CRISPR Screens. In *Genome Biology*. doi:10.1186/s13059-020-01972-x

[18] Henrique Botelho Guerra, João F. Ferreira, and João Costa Seco. 2023. Hoogle★: Constants and λ-Abstractions in Petri-net-based Synthesis Using Symbolic Execution. In *37th European Conference on Object-Oriented Programming (ECOOP)*. doi:10.4230/LIPIcs.ECOOP.2023.4

[19] Martin Bravenboer and Yannis Smaragdakis. 2009. Strictly Declarative Specification of Sophisticated Points-to Analyses. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*. doi:10.1145/1640089.1640108

[20] Nicolas L. Bray, Harold Pimentel, Páll Melsted, and Lior Pachter. 2016. Near-Optimal Probabilistic RNA-seq Quantification. In *Nature Biotechnology*. doi:10.1038/nbt.3519

[21] Yanju Chen, Chenglong Wang, Xinyu Wang, Osbert Bastani, and Yu Feng. 2023. Fast and Reliable Program Synthesis via User Interaction. In *38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. doi:10.1109/ASE56229.2023.00129

[22] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing Database-Backed Applications with Query Synthesis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2491956.2462180

[23] Richard H. Connelly and F. Lockwood Morris. 1995. A Generalization of the Trie Data Structure. In *Mathematical Structures in Computer Science*. doi:10.1017/S0960129500000803

[24] Matthew Conway, Steve Audia, Tommy Burnette, Dennis Cosgrove, and Kevin Christiansen. 2000. Alice: Lessons Learned from Building a 3D System for Novices. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. doi:10.1145/332040.332481

[25] Timothy P. Daley, Zhixiang Lin, Xueqiu Lin, Yanxia Liu, Wing Hung Wong, and Lei S. Qi. 2018. CRISPhieRmix: A Hierarchical Mixture Model for CRISPR Pooled Screens. In *Genome Biology*. doi:10.1186/s13059-018-1538-6

[26] Aaron A. Diaz, Han Qin, Miguel Ramalho-Santos, and Jun S. Song. 2015. HiTSelect: A Comprehensive Tool for High-Complexity-Pooled Screen Analysis. In *Nucleic Acids Research*. doi:10.1093/nar/gku1197

[27] Alexander Dobin, Carrie A. Davis, Felix Schlesinger, Jorg Drenkow, Chris Zaleski, Sonali Jha, Philippe Batut, Mark Chaisson, and Thomas R. Gingeras. 2013. STAR: Ultrafast Universal RNA-seq Aligner. In *Bioinformatics*. doi:10.1093/bioinformatics/bts635

[28] Azadeh Farzan, Danya Lette, and Victor Nicolet. 2022. Recursion Synthesis with Unrealizability Witnesses. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3519939.3523726

[29] Azadeh Farzan and Victor Nicolet. 2021. Counterexample-Guided Partial Bounding for Recursive Function Synthesis. In *Computer Aided Verification (CAV)*. doi:10.1007/978-3-030-81685-8_39

[30] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3192366.3192382

[31] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-Based Synthesis of Table Consolidation and Transformation Tasks from Examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3062341.3062351

[32] Yu Feng, Ruben Martins, Yuepeng Wang, Isil Dillig, and Thomas W. Reps. 2017. Component-Based Synthesis for Complex APIs. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/3009837.3009851

[33] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Programming Languages and Systems (APLAS)*. doi:10.1007/978-3-319-26529-2_25

[34] John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2737924.2737977

[35] David B. Garlan and Philip L. Miller. 1984. GNOME: An Introductory Programming Environment Based on a Family of Structure Editors. In *ACM SIGPLAN Notices*. doi:10.1145/390011.808250

[36] Gerhard Gentzen. 1964. Investigations into Logical Deduction. In *American Philosophical Quarterly*. jstor:20009142

[37] Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/1926385.1926423

[38] Sumit Gulwani, Oleksandr Polozov, and Rishabh Singh. 2017. *Program Synthesis*. doi:10.1561/2500000010

[39] Zheng Guo, Michael James, David Justo, Jiaxiao Zhou, Ziteng Wang, Ranjit Jhala, and Nadia Polikarpova. 2019. Program Synthesis by Type-Guided Abstraction Refinement. In *Proceedings of the ACM on Programming Languages, Issue POPL*. doi:10.1145/3371080

[40] Traver Hart and Jason Moffat. 2016. BAGEL: A Computational Framework for Identifying Essential Genes from Pooled Library Screens. In *BMC Bioinformatics*. doi:10.1186/s12859-016-1015-8

[41] Brian Hempel, Justin Lubin, Grace Lu, and Ravi Chugh. 2018. Deuce: A Lightweight User Interface for Structured Editing. In *Proceedings of the 40th International Conference on Software Engineering (ICSE)*. doi:10.1145/3180155.3180165

[42] Martin Hofmann and Emanuel Kitzelmann. 2010. I/O Guided Detection of List Catamorphisms: Towards Problem Specific Use of Program Templates in IP. In *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM)*. doi:10.1145/1706356.1706375

[43] Qiantan Hong and Alex Aiken. 2024. Recursive Program Synthesis Using Paramorphisms. In *Proceedings of the ACM on Programming Languages, Issue PLDI*. doi:10.1145/3656381

[44] Jingmei Hu, Priyan Vaithilingam, Stephen Chong, Margo Seltzer, and Elena L. Glassman. 2021. Assuage: Assembly Synthesis Using A Guided Exploration. In *The 34th Annual ACM Symposium on User Interface Software and Technology (UIST)*. doi:10.1145/3472749.3474740

[45] Michael B. James, Zheng Guo, Ziteng Wang, Shivani Doshi, Hila Peleg, Ranjit Jhala, and Nadia Polikarpova. 2020. Digging for Fold: Synthesis-Aided API Discovery for Haskell. In *Proceedings of the ACM on Programming Languages, Issue OOPSLA*. doi:10.1145/3428273

[46] Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *ACM/IEEE 32nd International Conference on Software Engineering (ICSE)*. doi:10.1145/1806799.1806833

[47] Ruyi Ji, Jingjing Liang, Yingfei Xiong, Lu Zhang, and Zhenjiang Hu. 2020. Question Selection for Interactive Program Synthesis. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3385412.3386025

[48] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. 2016. Verified Lifting of Stencil Computations. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2908080.2908117

[49] Etienne Kneuss, Ivan Kuraj, Viktor Kuncak, and Philippe Suter. 2013. Synthesis Modulo Recursive Functions. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*. doi:10.1145/2509136.2509555

[50] Tristan Knoth, Di Wang, Nadia Polikarpova, and Jan Hoffmann. 2019. Resource-Guided Program Synthesis. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/3314221.3314602

[51] Amy J. Ko and Brad A. Myers. 2006. Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI)*. doi:10.1145/1124772.1124831

[52] Renate König, Chih-yuan Chiang, Buu P. Tu, S. Frank Yan, Paul D. DeJesus, Angelica Romero, Tobias Bergauer, Anthony Orth, Ute Krueger, Yingyao Zhou, and Sumit K. Chanda. 2007. A Probability-Based Approach for the Analysis of Large-Scale RNAi Screens. In *Nature Methods*. doi:10.1038/nmeth1089

[53] James Koppel, Zheng Guo, Edsko de Vries, Armando Solar-Lezama, and Nadia Polikarpova. 2022. Searching Entangled Program Spaces. In *Proceedings of the ACM on Programming Languages, Issue ICFP*. doi:10.1145/3547622

[54] Ben Langmead and Steven L. Salzberg. 2012. Fast Gapped-Read Alignment with Bowtie 2. In *Nature Methods*. doi:10.1038/nmeth.1923

[55] Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. 2017. Interactive Program Synthesis. arXiv:1703.03539 [cs]

[56] Woosuk Lee and Hangyeol Cho. 2023. Inductive Synthesis of Structurally Recursive Functional Programs from Non-recursive Expressions. In *Proceedings of the ACM on Programming Languages, Issue POPL*. doi:10.1145/3571263

[57] Wei Li, Johannes Köster, Han Xu, Chen-Hao Chen, Tengfei Xiao, Jun S. Liu, Myles Brown, and X. Shirley Liu. 2015. Quality Control, Modeling, and Visualization of CRISPR Screens with MAGeCK-VISPR. In *Genome Biology*. doi:10.1186/s13059-015-0843-6

[58] Wei Li, Han Xu, Tengfei Xiao, Le Cong, Michael I. Love, Feng Zhang, Rafael A. Irizarry, Jun S. Liu, Myles Brown, and X. Shirley Liu. 2014. MAGeCK Enables Robust Identification of Essential Genes from Genome-Scale CRISPR/Cas9 Knockout Screens. In *Genome Biology*. doi:10.1186/s13059-014-0554-4

[59] Yang Liao, Gordon K. Smyth, and Wei Shi. 2014. featureCounts: An Efficient General Purpose Program for Assigning Sequence Reads to Genomic Features. In *Bioinformatics*. doi:10.1093/bioinformatics/btt656

[60] M. H. Löb. 1976. Embedding First Order Predicate Logic in Fragments of Intuitionistic Logic. In *The Journal of Symbolic Logic*. doi:10.2307/2272390

[61] Michael I. Love, Wolfgang Huber, and Simon Anders. 2014. Moderated Estimation of Fold Change and Dispersion for RNA-seq Data with DESeq2. In *Genome Biology*. doi:10.1186/s13059-014-0550-8

[62] Justin Lubin, Nick Collins, Cyrus Omar, and Ravi Chugh. 2020. Program Sketching with Live Bidirectional Evaluation. In *Proceedings of the ACM on Programming Languages, Issue ICFP*. doi:10.1145/3408991

[63] Justin Lubin, Jeremy Ferguson, Kevin Ye, Jacob Yim, and Sarah E. Chasins. 2024. Equivalence by Canonicalization for Synthesis-Backed Refactoring. In *Proceedings of the ACM on Programming Languages, Issue PLDI*. doi:10.1145/3656453

[64] Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Honeybee Docker Image. Zenodo. doi:10.5281/zenodo.15033531

[65] Justin Lubin, Parker Ziegler, and Sarah E. Chasins. 2025. Honeybee Docker Image (Exact Version for Artifact Evaluation). Zenodo. doi:10.5281/zenodo.15047790

[66] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. 2016. From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2908080.2908096

[67] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. 2005. Jungloid Mining: Helping to Navigate the API Jungle. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/1065010.1065018

[68] Benjamin Mariano, Yanju Chen, Yu Feng, Greg Durrett, and Işil Dillig. 2022. Automated Transpilation of Imperative to Functional Code Using Neural-Guided Program Synthesis. In *Proceedings of the ACM on Programming Languages, Issue OOPSLA1*. doi:10.1145/3527315

[69] Jeffrey G. Masek, Michael A. Wulder, Brian Markham, Joel McCorkel, Christopher J. Crawford, James Storey, and Del T. Jenstrom. 2020. Landsat 9: Empowering open science and applications through continuity. In *Remote Sensing of Environment*. doi:10.1016/j.rse.2020.111968

[70] Mikaël Mayer, Gustavo Soares, Maxim Grechkin, Vu Le, Mark Marron, Oleksandr Polozov, Rishabh Singh, Benjamin Zorn, and Sumit Gulwani. 2015. User Interaction Models for Disambiguation in Programming by Example. In *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology (UIST)*. doi:10.1145/2807442.2807459

[71] Robin M. Meyers, Jordan G. Bryan, James M. McFarland, Barbara A. Weir, Ann E. Sizemore, Han Xu, Neekesh V. Dharia, Phillip G. Montgomery, Glenn S. Cowley, Sasha Pantel, Amy Goodale, Yenarae Lee, Levi D. Ali, Guozhi Jiang, Rakela Lubonja, William F. Harrington, Matthew Strickland, Ting Wu, Derek C. Hawes, Victor A. Zhivich, Meghan R. Wyatt, Zohra Kalani, Jaime J. Chang, Michael Okamoto, Kimberly Stegmaier, Todd R. Golub, Jesse S. Boehm, Francisca Vazquez, David E. Root, William C. Hahn, and Aviad Tsherniak. 2017. Computational Correction of Copy Number Effect Improves Specificity of CRISPR–Cas9 Essentiality Screens in Cancer Cells. In *Nature Genetics*. doi:10.1038/ng.3984

[72] Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. In *Proceedings of the ACM on Programming Languages, Issue POPL*. doi:10.1145/3498682

[73] Leonardo de Moura and Sebastian Ullrich. 2021. The Lean 4 Theorem Prover and Programming Language. In *Automated Deduction (CADE)*. doi:10.1007/978-3-030-79876-5_37

[74] Niek Mulleners, Johan Jeuring, and Bastiaan Heeren. 2023. Program Synthesis Using Example Propagation. In *Practical Aspects of Declarative Languages (PADL)*. doi:10.1007/978-3-031-24841-2_2

[75] Daye Nam, Baishakhi Ray, Seohyun Kim, Xianshan Qu, and Satish Chandra. 2022. Predictive Synthesis of API-centric Code. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS)*. doi:10.1145/3520312.3534866

[76] Ulf Norell. 2007. *Towards a Practical Programming Language Based on Dependent Type Theory*. Ph. D. Dissertation. Department of Computer Science and Engineering, Chalmers University of Technology.

[77] Vegard Nygaard, Einar Andreas Rødland, and Eivind Hovig. 2016. Methods That Remove Batch Effects While Retaining Group Differences May Lead to Exaggerated Confidence in Downstream Analyses. In *Biostatistics*. doi:10.1093/biostatistics/kxv027

[78] Cyrus Omar, Ian Voysey, Michael Hilton, Jonathan Aldrich, and Matthew A. Hammer. 2017. Hazelnut: A Bidirectionally Typed Structure Editor Calculus. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. doi:10.1145/3009837.3009900

[79] Peter-Michael Osera and Steve Zdancewic. 2015. Type-and-Example-Directed Program Synthesis. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2737924.2738007

[80] Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: A Framework for Synthesizing Data Profiles. In *Proceedings of the ACM on Programming Languages, Issue OOPSLA*. doi:10.1145/3276520

[81] Rob Patro, Geet Duggal, Michael I. Love, Rafael A. Irizarry, and Carl Kingsford. 2017. Salmon Provides Fast and Bias-Aware Quantification of Transcript Expression. In *Nature Methods*. doi:10.1038/nmeth.4197

[82] Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. 2020. Programming with a Read-Eval-Synth Loop. *Proceedings of the ACM on Programming Languages, Issue OOPSLA* (Nov. 2020). doi:10.1145/3428227

[83] Hila Peleg, Shachar Itzhaky, and Sharon Shoham. 2018. Abstraction-Based Interaction Model for Synthesis. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*. doi:10.1007/978-3-319-73721-8_18

[84] Hila Peleg, Sharon Shoham, and Eran Yahav. 2018. Programming Not Only by Example. In *IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. doi:10.1145/3180155.3180189

[85] Nadia Polikarpova, Ivan Kuraj, and Armando Solar-Lezama. 2016. Program Synthesis from Polymorphic Refinement Types. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2908080.2908093

[86] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. 2019. Provenance-Guided Synthesis of Datalog Programs. In *Proceedings of the ACM on Programming Languages, Issue POPL*. doi:10.1145/3371130

[87] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay Silver, Brian Silverman, and Yasmin Kafai. 2009. Scratch: Programming for All. In *Communications of the ACM (CACM)*. doi:10.1145/1592761.1592779

[88] The Rocq Development Team. 2025. The Rocq Prover. Zenodo. doi:10.5281/zenodo.15149629

[89] J. W. Rouse, R. H. Haas, J. A. Schell, and D. W. Deering. 1974. *Monitoring Vegetation Systems in the Great Plains with ERTS.* Technical Report. NASA. https://ntrs.nasa.gov/citations/19740022614

[90] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till Westmann. 2016. On Fast Large-Scale Program Analysis in Datalog. In *Proceedings of the 25th International Conference on Compiler Construction (CC)*. doi:10.1145/2892208.2892226

[91] Ophir Shalem, Neville E. Sanjana, Ella Hartenian, Xi Shi, David A. Scott, Tarjei S. Mikkelsen, Dirk Heckl, Benjamin L. Ebert, David E. Root, John G. Doench, and Feng Zhang. 2014. Genome-Scale CRISPR-Cas9 Knockout Screening in Human Cells. In *Science*. doi:10.1126/science.1247005

[92] Kensen Shi, David Bieber, and Rishabh Singh. 2022. TF-Coder: Program Synthesis for Tensor Manipulations. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*. doi:10.1145/3517034

[93] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. 2018. Syntax-Guided Synthesis of Datalog Programs. In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. doi:10.1145/3236024.3236034

[94] Calvin Smith and Aws Albarghouthi. 2016. MapReduce Program Synthesis. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/2908080.2908102

[95] Armando Solar-Lezama, Christopher Grant Jones, and Rastislav Bodik. 2008. Sketching Concurrent Data Structures. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. doi:10.1145/1375581.1375599

[96] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. 2006. Combinatorial Sketching for Finite Programs. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/1168857.1168907

[97] Jerry Swan, Krzysztof Krawiec, and Zoltan A. Kocsis. 2019. Stochastic Synthesis of Recursive Functions Made Easy with Bananas, Lenses, Envelopes and Barbed Wire. In *Genetic Programming and Evolvable Machines*. doi:10.1007/s10710-019-09347-3

[98] Tamás Szabó, Gábor Bergmann, Sebastian Erdweg, and Markus Voelter. 2018. Incrementalizing Lattice-Based Program Analyses in Datalog. In *Proceedings of the ACM on Programming Languages, Volume 2, Issue OOPSLA*. doi:10.1145/3276509

[99] Tim Teitelbaum and Thomas Reps. 1981. The Cornell Program Synthesizer: A Syntax-Directed Programming Environment. In *Communications of the ACM (CACM)*. doi:10.1145/358746.358755

[100] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. 2017. Interactive Query Synthesis from Input-Output Examples. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD)*. doi:10.1145/3035918.3058738

[101] Tim Wang, Jenny J. Wei, David M. Sabatini, and Eric S. Lander. 2014. Genetic Screens in Human Cells Using the CRISPR-Cas9 System. In *Science*. doi:10.1126/science.1246981

[102] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration Using Datalog Program Synthesis. In *Proceedings of the VLDB Endowment*. doi:10.14778/3384345.3384350

[103] John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. 2005. Using Datalog with Binary Decision Diagrams for Program Analysis. In *Programming Languages and Systems (APLAS)*. doi:10.1007/11575467_8

[104] A. K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. In *Information and Computation*. doi:10.1006/inco.1994.1093

[105] Yongwei Yuan, Arjun Radhakrishna, and Roopsha Samanta. 2023. Trace-Guided Inductive Synthesis of Recursive Functional Programs. In *Proceedings of the ACM on Programming Languages, Issue PLDI*. doi:10.1145/3591255

[106] Tianyi Zhang, Zhiyang Chen, Yuanli Zhu, Priyan Vaithilingam, Xinyu Wang, and Elena L. Glassman. 2021. Interpretable Program Synthesis. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems (CHI)*. doi:10.1145/3411764.3445646

[107] Tianyi Zhang, London Lowmanstone, Xinyu Wang, and Elena L. Glassman. 2020. Interactive Program Synthesis by Augmented Examples. In *Proceedings of the 33rd Annual ACM Symposium on User Interface Software and Technology (UIST)*. doi:10.1145/3379337.3415900

[108] Yuqing Zhang, Giovanni Parmigiani, and W. Evan Johnson. 2020. ComBat-seq: Batch Effect Adjustment for RNA-seq Count Data. In *NAR Genomics and Bioinformatics*. doi:10.1093/nargab/lqaa078

[109] Yihong Zhang, Yisu Remy Wang, Oliver Flatt, David Cao, Philip Zucker, Eli Rosenthal, Zachary Tatlock, and Max Willsey. 2023. Better Together: Unifying Datalog and Equality Saturation. In *Proceedings of the ACM on Programming Languages, Issue PLDI*. doi:10.1145/3591239

[110] Zhanhui Zhou, Man To Tang, Qiping Pan, Shangyin Tan, Xinyu Wang, and Tianyi Zhang. 2022. INTENT: Interactive Tensor Transformation Synthesis. In *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology (UIST)*. doi:10.1145/3526113.3545653

Received 2024-11-15; accepted 2025-03-06