Eliciting Domain Expertise Reduces Examples Needed for Program Synthesis

Jeremy Ferguson

Electrical Engineering and Computer Sciences
University of California, Berkeley

Technical Report No. UCB/EECS-2024-83
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2024/EECS-2024-83.html

May 10, 2024

Acknowledgement

I would like to first thank my advisor, Professor Sarah E. Chasins, whose support and insight was crucial to the design and implementation of this project. Your expertise, wisdom and encouragement were incredibly helpful and I greatly appreciate all the work you put in.

I would also like to thank my mentor, Justin Lubin, who has provided so much support over the last two and a half years and especially over the course of this project. You have helped me to grow in my research capabilities and I am so grateful for all of the work you have done over the years to mentor and guide me.

Eliciting Domain Expertise Reduces Examples Needed for Program Synthesis
by Jeremy Ferguson

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

[illegible]

Professor Chasins
Research Advisor

May 12, 2024

* * * * * * *

Professor Parameswaran
Second Reader

May 12, 2024

bConnected powered by Google

Jeremy Ferguson <jmfergie@berkeley.edu>

Faculty Reader Request

Aditya Parameswaran <adityagp@berkeley.edu> Fri, May 10, 2024 at 4:39 PM
To: Jeremy Ferguson <jmfergie@berkeley.edu>

Hi Jeremy, I'm on the road and out of town and I can't sign this until tonight. You can use this email as approval in the meantime.

[Quoted text hidden]

Eliciting Domain Expertise Reduces Examples Needed for Program Synthesis

While Programming By Example (PBE) holds the promise of putting programming tasks in reach for non-programmer domain experts, PBE for complex programs often requires a large number of examples. We aim to bring the benefits of PBE to domains in which providing sufficient examples is prohibitively tedious or time-consuming. We build our approach around the insight that users often have useful domain knowledge that input-output examples can express only indirectly. We introduce a synthesis framework that accepts as input both: (i) traditional labeled examples and (ii) decisions about whether particular domain concepts are relevant to the task at hand. This novel synthesis framework allows us to explore the tradeoff between providing examples vs. domain knowledge. For a sample domain, we find that a difference of 8 additional domain concept decisions improves synthesized programs as much as 80 additional input-output examples. To our knowledge, this is the first synthesis approach that brings together examples and explicitly elicited and checked domain knowledge; we hope it opens the door to additional approaches for this new flavor of multi-modal program synthesis.

1 INTRODUCTION

For decades, we have known that users of programming-by-example (PBE) systems find it frustrating to provide many repetitive examples in cases where they feel they could express their domain expertise more directly. Indeed, Lau [2009] writes that "Unlike most machine-learning systems that can rely on hundreds or thousands of training examples, users are rarely willing to provide more than a handful of examples from which [a program synthesizer] can generalize," noting also that "users often have knowledge about their task that is not known to the system."

Typically, PBE research answers the frustrations of providing many repetitive input-output examples by guiding the choice of examples to those that would be most helpful for the synthesizer [Jha et al., 2010, Padhi et al., 2018, Ji et al., 2020, Laich et al., 2020, Chen et al., 2023]. The goal is to reduce example redundancy and thus ultimately reduce the number of examples required.

In this work we take an alternative approach built on the observation that a synthesis user may have domain knowledge about the task at hand that they can share explicitly, rather than only indirectly via examples. As Lau observed, PBE users often complain that they want to provide direct feedback on the domain and the system's hypotheses "without having to retrain the system with additional examples" [Lau, 2009]. We propose a synthesis framework that builds this kind of feedback into the synthesis interaction model. Our synthesis algorithm suggests domain concepts that may or may not be relevant to the task at hand—we use these decisions about domain concepts as an input to the synthesis algorithm, in addition to traditional PBE-style examples.

Introducing a synthesis approach that accepts both examples and domain knowledge allows us to explore how much synthesis benefits from each. In particular, it gives us a tool to interrogate the intuition that one domain concept is more informative, on average, than one labeled example. Traditional PBE approaches for reducing example annotation burdens ask: What strategy should we use for requesting examples in order to reduce number of examples required overall? In contrast, we design our synthesis approach so that we can ask:

Can we reduce the number of examples required for synthesis by querying domain expertise?

Section 2 provides an overview of how our synthesis algorithm solicits information about domain concepts. To create a program synthesis algorithm that lets us pose our animating question, we

Author's address:

need a definition of what it means for a synthesized program to respect domain expertise; Section 3 provides this definition and a synthesis framework built around the definition. In Section 4, we instantiate our synthesis framework for a specific problem domain, scoring documents (e.g. images) on the basis of extracted features. This instantiation generates programs in a language of scores guarded by predicates similar to Awk [Aho et al., 1979]. For example, our synthesizer could generate the following program using features extracted from a computer vision model to find images of people playing baseball or volleyball:

$$\{\ \exists \mathsf{person} \land \exists \mathsf{baseball} \rightarrow +1,\ \ \exists \mathsf{person} \land \exists \mathsf{volleyball} \rightarrow +1\ \},$$

the semantics of which when applied to an image is to sum up scores from all the predicates that are satisfied. Synthesized programs could directly be compiled to standard Python programs that make calls to the underlying black-box functions that produce these features, such as computer vision models or any other tool that can generate user-interpretable features for the documents of interest.

In Section 5, we demonstrate empirically how our instantiation behaves on a set of tasks; in particular, we do observe a tradeoff between oracle-labeled examples and explicit domain expertise. We note that we do not include a user study in our evaluation, as our goal is to assess whether the folklore hypothesis that domain expertise can can substitute for examples actually holds in a real synthesizer.

In summary, this report makes the following contributions:

(1) A program synthesis framework in which the specification is a combination of (i) examples (as in traditional PBE) and (ii) decisions about whether particular domain concepts are relevant to the target task.
(2) An algorithm for suggesting promising domain concepts based on the specification provided so far.
(3) An implementation of our synthesis approach called DExScore and an empirical evaluation of it that assesses, among other questions, (i) whether we can substitute domain concept decisions for additional input-output examples and achieve the same program performance, (ii) the effectiveness of the concept suggestion algorithm.

2 OVERVIEW

Motivating Example: Image Selection Task. Consider the task of determining whether a large set of images includes people playing sports or does not include people playing sports. Although this example may appear simple, it provides a good testing ground to explore questions about how domain knowledge can be explicitly incorporated and modeled into a synthesizer. Moreover, this task is representative of a general category of tasks that aim to assign scores (or classifications) in a structured way, such as determining whether crosswalks are safe for pedestrians or parks meet accessibility requirements.

In our task, a user might have some domain knowledge about what makes an image a sports-related image, but they need a way to turn that into a system that can do the classification rather than having to manually label each image. In this section, we consider possible approaches to this problem including traditional programming, using a large language model (LLM), and Programming By Example (PBE), before ultimately presenting our approach to directly leverage domain knowledge.

Other Approaches. Traditionally, to identify the images of interest in our motivating example task, the user would need to write a program, directly encoding their domain knowledge. The downside of this approach is that it requires recall rather than recognition, which has long been known to be more challenging [MacDougall, 1904]. Much domain knowledge is implicit—e.g., the user may not be able to state that they are looking for the presence of soccer balls, hockey sticks, volleyballs, frisbees, goalposts, surfboards, even if they would be able to confirm the relevance of those items if asked. This may make it difficult for users to operationalize their domain knowledge in the form of a program. This approach also puts automation out of reach for domain experts who are not comfortable programming.

More recently, as LLMs become more viable for a broad range of tasks, the user could write a system that prompts an LLM to perform the classification of each image, instructing it about the task in natural language. The downsides to this approach are: (1) There is no program artifact for the programmer to audit or tweak; they must trust the LLM to interpret their instructions correctly on each new image. (2) Further, the LLM-based approach works only for tasks for which the LLM itself has the relevant domain expertise. If the user aims to complete a task that requires niche or specialized expertise (unlike the sports example), relying on an LLM to do the work of producing the implicit domain knowledge is a non-starter. This constrains the LLM approach to common knowledge-style tasks. This is especially an obstacle if the user wants to represent knowledge about their own particular preferences or taste (e.g., "find images I like" as opposed to "find images that depict a person playing sports"). As with traditional programming, expressing these subjective preferences and tastes requires the user to do the difficult work of recall instead of recognition. An alternative would be to ask the LLM for a program that automates the task. This addresses issue (1) but not issue (2). This also introduces a new obstacle, obstacle (3), that is specific to non-programmers and novice programmers; prior work has demonstrated that novice programmers struggle to use LLMs for writing programs [Nguyen et al., 2024].

We aim to build a system that relies only on recognition of relevant domain concepts, rather than recall. Additionally, we want to produce a program that the user can read or extend after the fact. PBE is one traditional approach for producing programs in the face of recognition versus recall challenges. Producing the target output for a given input is one way of sharing information about implicit domain knowledge, a method that does not require operationalizing the knowledge in the form of a program or set of natural language instructions. However, PBE systems may require many repetitive examples even after the point at which a relevant domain concept has become clear.

Soliciting Decisions About Domain Concepts. Our system is built on the core idea of iteratively suggesting domain knowledge concepts to the user, allowing the user to approve or reject the domain concepts, then using these same concepts in the program itself. (See Figure 1.) In particular, to operationalize domain knowledge, we choose predicates over the synthesized program's inputs. For example, for the image task above, our framework might suggest a predicate that flags whether an image includes a depiction of both a person and a baseball: $\exists\texttt{person} \land \exists\texttt{baseball}$. If the task at hand was to query a database of music scores, a predicate might flag whether a piece includes a III chord immediately followed by IV and I: $\exists(\text{III}; \text{IV}; \text{I})$. The framework asks an oracle for a ruling on whether the suggested predicate is relevant. If yes, the predicate is incorporated into the synthesized program. If no, the predicate is discarded.

Intuition. Given this overall framework, the key challenge is to suggest likely-relevant domain concepts. Our suggestion algorithm uses:

(1) An Extract function which divides the input documents into two (possibly overlapping) subsets, X^+ and X^-. The implementation of Extract is relatively unconstrained, but intuitively X^+ should include documents that the synthesized program should accept, while X^- should include documents that the synthesized program should reject. Alternatively, X^- may constitute documents for which the system does not yet have a good prediction.

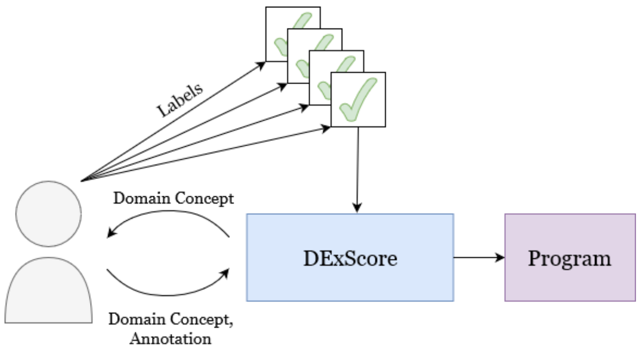

Fig. 1. **User interaction model.** The user provides an intial set of labeled examples. DExScore suggests domain concepts which may or may not be relevant to the task at hand. The user annotates the domain concepts to indicate whether they are relevant.

(2) A Ratio function which computes the log-likelihood ratio of each predicate between X^+ and X^-. That is, it applies each candidate predicate to each input document and produces a measure of whether the property holds more often in one subset or the other. The intuition here is that a predicate with a high log-likelihood ratio holds much more often on documents in X^+ than on documents in X^-, and therefore may represent domain knowledge that helps to distinguish documents that the synthesized program should accept from documents it should reject.

3 FRAMEWORK AND PROBLEM STATEMENT

As we are specifically interested in modeling domain expertise, we now formalize our problem statement via a series of definitions that capture this idea. Specifically, we consider the situation in which we aim to synthesize a program that, given an input, returns a *score* in $\mathbb{R}$. While restricting the output of programs to be a single number may seem limiting, we will see that we can use this formalize both to (i) generalize programming-by-example and ML-based classifiers as well as (ii) to model precisely what we mean by domain expertise.

Definition 3.1 (Preliminaries). A **partially-ordered set** (also called a **poset**) over a set S is a pair $(S, \preceq)$, where $\preceq$ forms a partial order over S. A function f between posets $(S_1, \preceq_1)$ and $(S_2, \preceq_2)$ is **monotonic** if $x \preceq_1 x'$ implies $f(x) \preceq_2 f(x')$ for all $x, x' \in S_1$.

Definition 3.2 (Domain expertise framework). Instantiations of our framework consist of

(1) A set of inputs $\mathcal{X}$;
(2) A class $\mathcal{D}$ of **domain knowledge elicitations** that are posets over $\mathcal{X}$;
(3) A set of programs Prog; and
(4) A semantics $[\![\cdot]\!]\,(\cdot) : \text{Prog} \times \mathcal{X} \to \mathbb{R}$ dictating how $P \in \text{Prog}$ assigns a **score** to each $x \in \mathcal{X}$.

Definition 3.3 (Correctness criterion). A program $P \in \text{Prog}$ **respects** a domain knowledge elicitation D (written $P \nearrow D$) if $[\![P]\!]\,(\cdot)$ is a monotone map from D to $\mathbb{R}$; that is, if

$$x \preceq_D x' \quad \text{implies} \quad [\![P]\!]\,(x) \leq [\![P]\!]\,(x').$$

Remark 3.4. Depending on the context, scores can represent different concepts, as we will see in the examples following this remark. The domain knowledge elicitation partial order represents an understanding of how domain objects compare to each other; our correctness criterion asserts that inputs higher in the domain knowledge elicitation partial order cannot be assigned lower scores.

1.

$$\begin{tikzcd}
 & \langle \text{cat, dog} \rangle & \\
\langle \text{cat} \rangle & & \langle \text{dog} \rangle \\
 & \langle \text{fox} \rangle &
\end{tikzcd}$$

2.

$$\begin{tikzcd}
 & \langle \text{cat, dog} \rangle & \\
\langle \text{cat} \rangle & & \langle \text{dog} \rangle \arrow[ll] \arrow[ul] \\
 & \langle \text{fox} \rangle \arrow[uu] \arrow[ul] &
\end{tikzcd}$$

3.

$$\begin{tikzcd}
 & \langle \text{cat, dog} \rangle & \\
\langle \text{cat} \rangle \arrow[ur] & & \langle \text{dog} \rangle \arrow[ul] \\
 & \langle \text{fox} \rangle \arrow[ul] \arrow[ur] &
\end{tikzcd}$$

4.

$$\begin{tikzcd}
 & \langle \text{cat, dog} \rangle & \\
\langle \text{cat} \rangle \arrow[ur] & & \langle \text{dog} \rangle \arrow[ul] \\
 & \langle \text{fox} \rangle &
\end{tikzcd}$$

Fig. 2. **Four domain knowledge elicitation posets in the instantiation of scoring documents.** Documents are elements of the posets, and their features are represented in angle brackets. The first poset represents no domain knowledge. The second poset represents the domain knowledge that "cat" alone is relevant. The third poset indicates "cat" and "dog" are both relevant. The fourth poset indicates "cat," "dog," and "fox" are all relevant.

$$\begin{tikzcd}
 & & \{2 \mapsto 4, 3 \mapsto 9\} & & \\
 & \{2 \mapsto 4\} \arrow[ur] & & \{3 \mapsto 9\} \arrow[ul] & \\
 & & \varnothing \arrow[ul] \arrow[ur] & & \\
\{5 \mapsto 6\} \arrow[urr] & \{2 \mapsto 4, 8 \mapsto 9\} \arrow[ur] & \{3 \mapsto 6\} \arrow[u] & \{8 \mapsto 8\} \arrow[ul] & \cdots \arrow[ull]
\end{tikzcd}$$

Fig. 3. **The programming-by-example domain knowledge elicitation poset.** The structure of this poset is completely forced by the choice of the partial function (input-output example) at its top.

Example 3.5 (Scoring documents). Consider the situation in which objects (like documents or locations) must be scored or ranked; for example, in assigning a pedestrian safety score to a crossroads or accessibility score to a park, or query tasks where documents are scored by relevance. We instantiate our framework to this setup formally in Section 4, but, briefly, inputs $x \in \mathcal{X}$ are documents with associated features from, say, a computer vision model, black-box library functions, or simply metadata (see Section 5.1 for a description of the tools we use for feature extraction). Taking the case specifically of querying documents, a domain knowledge elicitation describes how some documents are less relevant than other documents. Figure 2 illustrate some possible orders. A program P will respect these domain knowledge elicitations when it does not assign a lower score to a document that is higher in the partial order.

Example 3.6 (Programming by example). Traditional programming-by-example (PBE) systems can be described in this framework as follows. The inputs $\mathcal{X}$ are partial functions (i.e., sets of input-output examples). The class of domain knowledge elicitations is $\mathcal{D} = \{D_{\text{pf}} \mid \text{pf is a partial function}\}$, where D_{pf} is the poset over $\mathcal{X}$ with the partial order $\preceq_{\text{pf}}$ such that $\text{pf}_1 \prec_{\text{pf}} \text{pf}_2$ (strict ordering) if $\text{pf}_2 \subseteq \text{pf}$ and $\text{pf}_2 \not\subseteq \text{pf}_1$. For programs P and partial functions pf, the semantics $[\![P]\!]\,(\text{pf})$ is 1 if P is an extension of pf (the traditional notion of example satisfaction in PBE), and 0 otherwise. Then our correctness criterion holds ($P \nearrow D_{\text{pf}}$) if and only if the traditional PBE correctness criterion holds ($[\![P]\!]\,(\text{pf}) = 1$).[1]

Figure 3 shows the structure of the PBE poset D_{pf} for an example partial function pf. Intuitively, the PBE poset does *not* capture much domain knowledge, as its structure is completely determine by pf; the PBE poset cannot capture more intricate relations between different elements.

Example 3.7 (ML-based classifiers). Traditional ML-based classifiers can be described in this framework as follows. The inputs $\mathcal{X}$ are elements to be classified (e.g., documents, images, etc.), and domain knowledge elicitations are *degenerate posets* (that is, $x \preceq x'$ if and only if $x = x'$). Programs P are classifiers with the semantics that for $x \in \mathcal{X}$, we have $[\![P]\!]\,(x) = c$ if P labels x as having class c (e.g., binary classifiers could label inputs into classes $\{-1, 1\}$). The correctness criterion in this case trivially holds *because no domain expertise whatsoever is encoded in the poset.* In this framework, labeled examples are not guaranteed to be classified correctly.

We now define what a program synthesizer is in our context; for our framework, synthesizers must both (i) elicit domain knowledge, and (ii) return programs that satisfy this domain knowledge.

Definition 3.8 (Synthesis). For our purposes, a **synthesizer** is an algorithm $\textsc{Synth} : \text{Spec} \longrightarrow \text{Prog} \times \mathcal{D}$ for some set of possible initial intent specifications Spec; crucially, the algorithm *may also make additional queries to an oracle in its execution.* Such a synthesizer is **sound** if for all $s \in \text{Spec}$, $\textsc{Synth}(s) = P, D$ implies $P \nearrow D$.[2]

Example 3.9. Building on Example 3.6, we can represent a traditional programming-by-example algorithm A that takes in a partial function as input-output examples as a synthesizer in our framework as $\textsc{Synth}(\text{pf}) = (A(\text{pf}), D_{\text{pf}})$. Similarly, based on Example 3.7, we can represent a traditional ML-based classifier C that takes in an element $x \in \mathcal{X}$ as $\textsc{Synth}(x) = (C(x), \mathcal{X}')$, where $\mathcal{X}'$ is the degenerate poset over $\mathcal{X}$.

We are particularly interested in the case in which the synthesizer takes input-output examples and elicits domain knowledge about which domain features should affect the scoring of domain inputs. This setup lets us investigate the tradeoff between input-output examples and domain expertise. Our notion of sound synthesizers in Definition 3.8 applied to this setup thus serve as solutions to our problem statement, which is as follows:

*Definition 3.10 (**Problem statement**).* Given (i) input-output examples and (ii) decisions about which domain features should affect scoring, return a program P and domain knowledge elicitation D that represents the gained knowledge such that $P \nearrow D$.

[1] *Proof.* If $P \nearrow$ pf, then, as $[\![P]\!]\,(\varnothing) = 1$ vacuously and $\varnothing \preceq_{\text{pf}}$ pf, we have $[\![P]\!]\,(\text{pf}) = 1$ by monotonicity. Conversely, if $[\![P]\!]\,(\text{pf}) = 1$, then for pf_1, pf_2 with $\text{pf}_1 \prec_{\text{pf}} \text{pf}_2$, we have $\text{pf}_2 \subseteq \text{pf}$ definitionally, so $[\![P]\!]\,(\text{pf}_2) = 1 \geq [\![P]\!]\,(\text{pf}_1)$. □

[2] We take a liberal view of what constitutes an "intent specification," including input-output examples or natural language specification, even those that do not fully "specify" a program. Our soundness criterion is thus with respect to the domain knowledge elicitation, not the initial specification.

$$\begin{array}{rl} \textbf{Programs} & \text{Prog} ::= \{p_i \rightarrow s_i\}^{i<L} \\ \textbf{Predicates} & p ::= \exists f \mid \neg p \mid p \wedge p \mid p \vee p \\ \textbf{Features} & f \in \text{Features} \\ \textbf{Scores} & s \in \mathbb{Z}^{\geq 0} \end{array}$$

Fig. 4. **The language of programs in our synthesis framework instantiation.**

4 FRAMEWORK INSTANTIATION AND SYNTHESIS ALGORITHM

We will now describe our instantiation of the framework outlined in Section 3.

4.1 Modeling Domain Expertise

As in Example 3.5, our inputs $x \in \mathcal{X}$ are documents with associated features from a fixed set of features Features. For a given document $x \in \mathcal{X}$, we let $x^{\#}$ be the set of all predicates that hold on x, where predicates are defined in Figure 4. We take a simple view of what predicates may be in Figure 4, as the exact choice is orthogonal to our synthesis algorithm; in particular, a predicate may be the existence of a feature f, and any combination of standard boolean operations (negation, conjunction, disjunction) thereof. We say the predicate $\exists f$ holds on a document x if f is in the features of x, and boolean operations are defined in the standard way.

The class of domain knowledge elicitations is $\mathcal{D} = \{D^{\star}_{Q} \mid Q \subseteq \text{Predicates}\}$, where $D^{\star}_{Q}$ is the poset over $\mathcal{X}$ with the partial order $\preceq_Q$ such that

$$x_1 \preceq_Q x_2 \quad \text{if} \quad Q \cap x_1^{\#} \subset Q \cap x_2^{\#},$$

where $\preceq_Q$ is not strict ordering, but $\subset$ is.

Intuitively, the predicate sets Q represent relevant predicates of the documents. For example, in classifying whether an image is a picture of sports, we might take $Q = \{\exists\text{baseball}, \exists\text{volleyball}\}$. Figure 2 gives examples of four such posets: $D^{\star}_{\varnothing}, D^{\star}_{\{\exists\text{cat}\}}, D^{\star}_{\{\exists\text{cat},\exists\text{dog}\}}$, and $D_{\{\exists\text{cat},\exists\text{dog},\exists\text{fox}\}}$. Note that the domain knowledge elicitation $D^{\star}_{\{\exists\text{cat}\}}$ marks the document $\langle\text{cat}\rangle$ as incomparable to the document $\langle\text{cat}, \text{dog}\rangle$, but $D_{\{\exists\text{cat},\exists\text{dog}\}}$ marks the document $\langle\text{cat}\rangle$ as less than or equal to the document $\langle\text{cat}, \text{dog}\rangle$, because when cat and dog are relevant, the latter document can be no less relevant than the former (which was not the case when just cat was relevant). This relation is captured precisely by the ordering $\preceq_Q$ defined above.

4.2 Language Syntax and Semantics

We can now define the set of programs in our instantiation. Figure 4 gives the syntax for our programs, which are sets of scores (non-negative integers) guarded by predicates, similar in spirit to the Awk [Aho et al., 1979] programming language. We define the semantics of a program P on an input x as follows:

$$[\![P]\!]\,(x) = \sum_{(p \rightarrow s) \in P} [p](x) \cdot s \quad \text{where} \quad [p](x) = \begin{cases} 1 & \text{if } p \text{ holds on } x \\ 0 & \text{otherwise} \end{cases}$$

4.3 Synthesis Algorithm

Given these semantics, we now present our synthesis algorithm in Figure 5. Our top-level synthesis algorithm, TopLevelSynth takes in as input an initial set of inputs marked as relevant and iteratively elicits domain knowledge and simultaneously constructs program that satisfies the tracked domain

```
**Hyperparameters:**
    $d$: The depth of predicates to search for
    $k$: The number of predicates to query the oracle with
**Inputs:**
    $X$: The starting set of relevant examples
    $P$: The current program
    $Q_{\text{sug}}$: The already-suggested predicates
1: function $\textsc{Suggest}(X, P, Q_{\text{sug}})$
2:     $X^+, X^- \leftarrow \textsc{Extract}(P, X)$
3:     $\Delta Q^0_{\text{sug}} \leftarrow \arg\max^k_{p \in \text{Predicates}_d \mid Q_{\text{sug}} \not\vdash p} |\textsc{Ratio}(p, X^+, X^-)|$
4:     $\Delta Q_{\text{sug}} \leftarrow \{p \text{ if } \textsc{Score}(p) > 0 \text{ else } \neg p \mid p \in \Delta Q^0_{\text{sug}}\}$
5:     return $\Delta Q_{\text{sug}}$
6: end function
```

```
**Inputs:**
    $X$: The starting set of relevant examples
    $P$: The current program
    $D$: The current domain knowledge elicitation
    $Q_{\text{sug}}$: The already-suggested predicates
1: function $\textsc{SynthStep}_X(P, D, Q_{\text{sug}})$
2:     $\Delta Q_{\text{sug}} \leftarrow \textsc{Suggest}(X, P, Q_{\text{sug}})$
3:     $\Delta P \leftarrow \{p \to 1 \text{ if oracle approves } p \mid p \in \Delta Q_{\text{sug}}\}$
4:     $P' \leftarrow P \cup \Delta P$
5:     $D' \leftarrow D^{\star}_{\text{dom}\, P'}$
6:     $Q'_{\text{sug}} \leftarrow Q_{\text{sug}} \cup \Delta Q_{\text{sug}}$
7:     return $P', D', Q'_{\text{sug}}$
8: end function
```

```
**Inputs:**
    $X$: The starting set of relevant examples
1: function $\textsc{TopLevelSynth}(X)$
2:     $P, D, Q_{\text{sug}} \leftarrow \varnothing, \varnothing, \varnothing$
3:     repeat
4:         $P, D, Q_{\text{sug}} \leftarrow \textsc{SynthStep}_X(P, D, Q_{\text{sug}})$
5:     until oracle terminates
6:     return $P, D$
7: end function
```

Fig. 5. Pseudocode for the synthesis functions. SynthStep represents one step in the synthesis loop. We separate our document space into X^+ and X^-, then use those subsets to compute the scores of all predicates, taking the top k predicates. We present those to the user, and add all accepted predicates to the program. TopLevelSynth is the entire synthesis program, which loops over synthesis steps until the user is satisfied with the final program.

knowledge elicitation. It does this by iterating the auxiliary function SynthStep, which relies on our predicate suggestion algorithm Suggest (which we describe below) to get predicates to query the oracle to make decisions about which ones are relevant. SynthStep takes the approved predicates and constructs a domain knowledge elicitation poset of the form described in Section 4.1 and a program that respects that domain knowledge elicitation. For simplicity, the right-hand side score values in our constructed program are always 1, but all soundness results apply with any non-negative score value; the score could be determined by, for example, querying the oracle.

Our predicate suggestion algorithm (Suggest) computes predicates as follows. First, using the auxiliary function Extract, it extracts two subsets from the set of inputs $\mathcal{X}$: a subset of programs that should likely be highly-scored (X^+), and a subset of programs that should likely be lowly-scored (X^-); we describe different implementations of Extract in Section 4.3.1 below. Then, for each possible predicate p up to fixed depth cutoff d that are not tautologically equivalent to an already-suggested predicate (notated $Q_{\text{sug}} \not\models p$), it calculates a log-likelihood ratio of p occurring on X^+ versus on X^- as follows:

$$\textsc{Ratio}(p, X^+, X^-) = \log_2 \frac{\mathbb{P}(p \mid X^+)}{\mathbb{P}(p \mid X^-)} = \log_2 \sum_{x \in X^+} \frac{[p](x)}{|X^+|} - \log_2 \sum_{x \in X^-} \frac{[p](x)}{|X^-|}.$$

Intuitively, predicates with a large positive or negative ratio correspond to highly discriminative predicates that we would want to present to the oracle. And indeed, Suggest concludes by taking the top k predicates by the absolute value of this ratio (notated with the superscript k on arg max), negating the predicate if its score is highly negative. We discuss a possible extension to this log-likelihood ratio calculation in Section 4.3.1 below.

We can now prove our synthesizer sound; our proof applies to any choice of how Extract and Ratio are computed.

Lemma 4.1. *For all $P \in$ Prog, $P \nearrow D^{\star}_{\operatorname{dom} P}$.*

Proof. Suppose $x_1, x_2 \in D^{\star}_{\operatorname{dom} P}$ with $x_1 \preceq_{\operatorname{dom} P} x_2$. Then

$$[\![P]\!]\,(x_1) = \sum_{(p \to s) \in P} [p](x_1) \cdot s = \sum_{\substack{(p \to s) \in P,\\ p \in x_1^\#}} s \leq \sum_{\substack{(p \to s) \in P,\\ p \in x_2^\#}} s = \sum_{(p \to s) \in P} [p](x_2) \cdot s = [\![P]\!]\,(x_2),$$

where the inequality holds because $\operatorname{dom} P \cap x_1^\# \subset \operatorname{dom} P \cap x_2^\#$ and scores s are non-negative. □

Corollary 4.2. *If $P \nearrow D$ and $\textsc{SynthStep}_{\mathcal{X}}(P, D, Q_{sug}) = P', D', Q'_{sug}$, then $P' \nearrow D'$.*

Proof. Immediate from Lemma 4.1 □

Theorem 4.3. *TopLevelSynth is sound; that is, if TopLevelSynth$(\mathcal{X}) = P, D$, then $P \nearrow D$.*

Proof. By induction, using Corollary 4.2. □

4.3.1 Variant Implementations of Extract and Ratio. In our evaluation, we investigate how sensitive our implementation of this synthesis algorithm is to various choices for computing Extract and Ratio; in brief, we find that our algorithm performs substantially better than suggesting random predicates for any of the variants we present here. We therefore keep our discussion of these variants relatively brief.

Extract Variants. A first choice for Extract—which we call the *direct* strategy—is to return $X^+ = X$ (the starting set of relevant inputs) and $X^- = \mathcal{X}$ (the entire input space); intuitively, the examples labeled as relevant should score higher than the entire input space as a whole. A second choice for Extract—which we call the *iterative* strategy—is to use the current program to guide the assignment of X^+ and X^-; inputs that score high or low on the current program get added to X^+ and X^- respectively. Mathematically, this choice is defined as:

$$\textsc{Extract}(P, X) = (X \cup \{x \in \mathcal{X} \mid \llbracket P \rrbracket (x) > t^+\}, \{x \in \mathcal{X} \mid \llbracket P \rrbracket (x) \leq t^-\}),$$

where t^+ and t^- are threshold hyperparameters ($t^+ = t^- = 1$ degrades to the direct strategy).

Ratio Variants. Once the log-likelihood ratios of predicates are computed, we can further filter the predicates to find those that, when combined, cover the largest subset of the document set using *mutual information* [Shannon, 1948]. Specifically, after finding the top k predicates by absolute log-likelihood ratio, we can select the set of k' predicates that have the lowest mutual information, where $k' < k$ is a hyperparameter ($k' = k$ degrades to just computing the log-likelihood ratios). We refer to this variant as the *mutual information* strategy.

4.4 Implementation

Our implementation of this instantiation, which we call DExScore, is freely available at [REDACTED].

5 EVALUATION

For our evaluation, we used DExScore (our instantiation of our broader framework, see Section 4) for sample tasks (i) that could be easily hand-labeled to produce a substantial ground-truth dataset, (ii) that could be accomplished with the kinds of features that a modern computer vision model can produce, and (iii) with which we have enough domain expertise to meaningfully interact with DExScore.

Our evaluation first explores our main research question:

RQ1. Does our approach enable trading off the number of PBE-style labeled examples for decisions based on domain expertise?

We also wanted to assess whether our algorithm to suggest promising domain concepts actually improved synthesis, so we investigate the following research question with an ablation study:

RQ2. How does our predicate suggestion strategy affect synthesis quality compared to random predicate suggestion?

Finally, we ask some additional questions to further probe DExScore:

RQ3. How do our synthesized output programs compare to using an LLM for the same task?
RQ4. How sensitive is our predicate suggestion algorithm to the choice of the Extract and Ratio functions?
RQ5. How does DExScore's execution time scale with program size?

5.1 Setup

For our sample tasks, we worked within the domain of images due to the wide availability of models for feature extraction on images. To extract features from the dataset of images, we used the DETR model, a transformer-based architecture which performs object detection [Carion et al., 2020]. Our goal is not to perform state-of-the-art image classification or to advance the state of the art in computer vision in any way; rather, our goal for this evaluation is to synthesize programs that use the results of an existing state-of-the-art computer vision model.

$$\begin{array}{rl}
\textbf{Programs} & \text{Prog} ::= \{p_i \longrightarrow s_i\}^{i<L} \\
\textbf{Predicates} & p ::= \exists f \mid \neg p \mid p \wedge p \mid p \vee p \mid \mathit{Eventually}(p,p) \mid \mathit{Nearby}(p,p) \\
\textbf{Features} & f \in \text{Features} \\
\textbf{Scores} & s \in \mathbb{Z}^{\geq 0}
\end{array}$$

Fig. 6. **The language of programs for the Music Task**

We ran all experiments on an Asus Vivobook Pro with an AMD Ryzen 9 7940HS running Windows 11 and Windows Subsystem for Linux 2 and Python 3.11.5.

5.1.1 Sports Task. For some components of the evaluation, we used a hand-labeled ground-truth image dataset; for these components, our task was "Is this an image of people playing sports?" (from this point on, Sports Task).

Sports Task Dataset. We used 2,427 images from the COCO dataset (test split), an image dataset with 91 different object classes commonly used for training and benchmarking state-of-the-art object detection models [Lin et al., 2015]. Before running DExScore or any computer vision algorithm, we hand-labeled our dataset with ground-truth labels, then ran the image through DETR to perform feature extraction.

5.1.2 Music Task. In addition to the Sports Task, we used a secondary task for one component of the evaluation, using a domain other than images. This task was "Is this piece of music similar to other pieces that have been selected by an oracle?" (from this point on, Music Task). This task can be viewed as a recommendation, querying, or preference elicitation task.

Music Task Dataset. We used a dataset consisting of the chord progressions of 52 Bach chorales, annotated by experts with 101 different chord labels [Radicioni and Esposito, 2014]. We performed an additional pre-processing step to capture temporal relationships within the chord progressions. We identified eight common cadences from music theory (small chord progressions of 2–5 chords that are commonly found within music pieces). We then annotated each chord progression with the location of each of these cadences. This annotation was done automatically. This list of cadences serves as the feature set. To create ground-truth labels for these documents, the first-author labeled them according to subjective preference.

Music Task DSL. For the Music Task, we augmented our DSL with additional predicates to capture the sequential nature of the chord progressions. In addition to the *And* and *Or* predicates, we added *Eventually* and *Nearby*. The *Eventually* predicate represents a sequential relationship "A then B". *Nearby* represents a sequential relationship "A then B, within n chords", where n is a fixed parameter, set to 20 for this evaluation. Figure 6 shows the syntax of this augmented DSL.

5.1.3 Oracles. Our synthesis algorithm relies on access to an oracle for soliciting domain expertise. To construct our oracles for the Sports and Music Tasks, we manually listed predicates relevant to our target question; the oracle accepted candidate predicates when they appeared in this list. For automatically-generated tasks (i.e., tasks in which the ground-truth program was automatically generated), we use an automatically-generated oracle that accepts predicates that appear in the ground-truth program and rejects predicates that do not appear in the ground-truth program.

5.1.4 Extract and Ratio Variants. For all experiments except the experiment explicitly probing the sensitivity of our approach to the choice of Extract and Ratio functions (Experiment 4), we

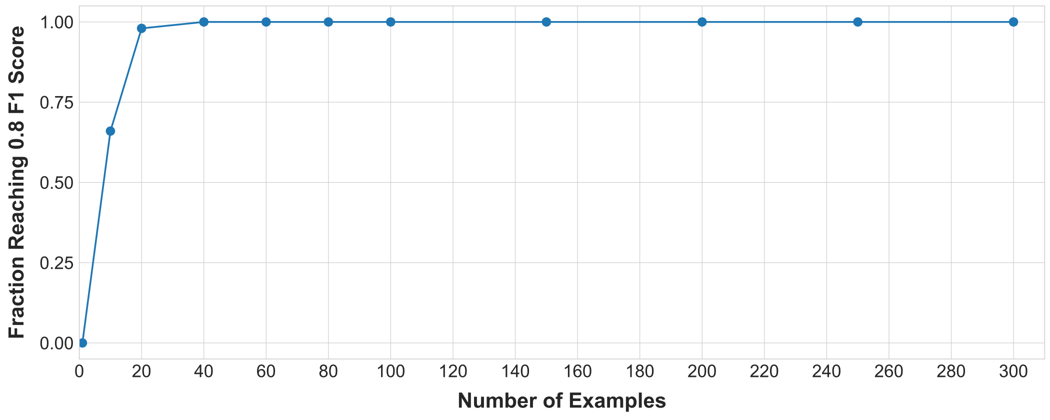

Fig. 7. **Fraction of runs that reach 0.8 F1 score.** Each run represents a different randomly-sampled subset of oracle-labeled documents provided as the initial input to the synthesizer. The position on the x-axis represents the number of labeled input documents provided to the synthesizer, and the position on the y-axis represents the fraction of runs for that number of documents that reached a 0.8 F1 score. When provided 20 or more examples, the synthesizer reaches the desired F1 score >95% of the time.

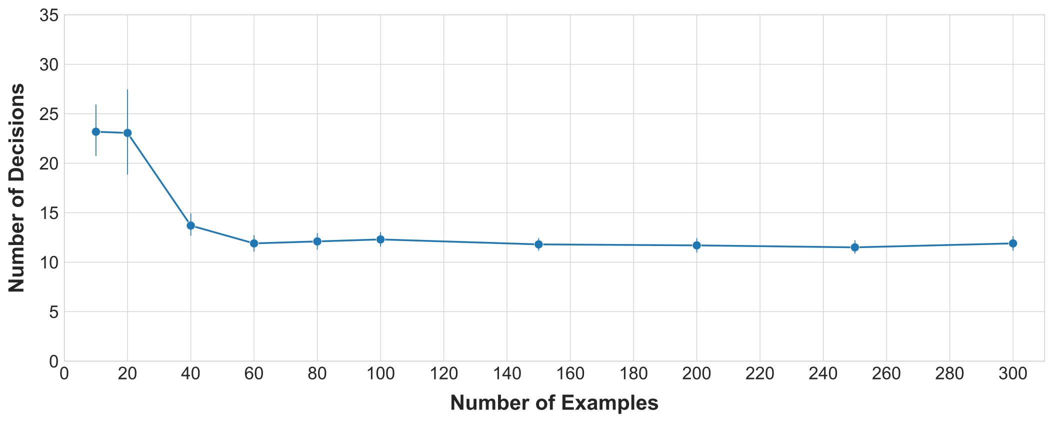

Fig. 8. **Tradeoff between example labeling work and domain expertise decision work.** Each point in the chart represents a set of synthesized programs that achieve the same F1 score of 0.8. The position on the x-axis represents the number of labeled input documents provided, and the position on the y-axis represents the average number of domain expertise decisions required to achieve the target F1 score (with 95% confidence interval error bars). To achieve the target F1 score, DExScore can take 23 labeled examples and 20 oracle decisions or 60 labeled examples and 12 oracle decisions; intuitively, $23 - 12 = 11$ domain concept decisions can substitute for $60 - 20 = 40$ labeled examples.

use the *iterative* strategy for Extract and the *mutual information strategy* for Ratio, as described in Section 4.3.1.

5.2 Experiment 1: Answering RQ1 (on Decisions vs. Examples Tradeoff)

5.2.1 Setup. We randomly sampled subsets of the labeled dataset of increasing size (50 sets per size) to serve as the initial set of relevant examples for DExScore. We then ran DExScore on each

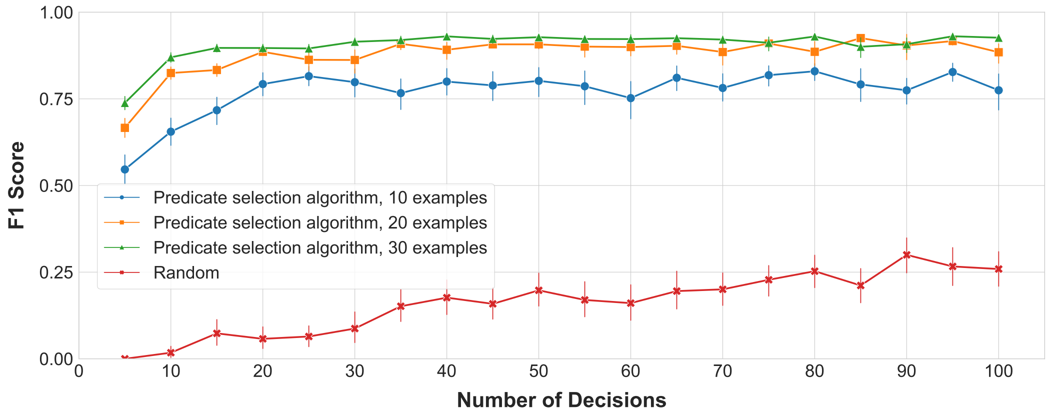

Fig. 9. **Predicate selection algorithm ablation.** This plot shows the performance of our predicate suggestion strategy, compared to a random suggestion strategy. The x-axis is the number of oracle decisions, and the y-axis is the F1 score. We provide DExScore with 10, 20, and 30 input examples, and report the F1 performance as oracle decisions vary (mean of 50 runs with 95% confidence interval error bars). We found that with all three example set sizes, our predicate suggestion algorithm outperforms a random suggestion strategy by a wide margin. For 20 oracle decisions or more, the mean F1 score of the example suggestion algorithm is above 0.75 for all three example set sizes, while the random strategy has a mean F1 score around 0.1–0.3.

of these initial specifications, terminating when either the generated program reached an F1 score of 0.8 (computed against our ground-truth labels) or the number of oracle decisions exceeded 150. We tracked (i) whether each of the 50 runs for each example suite size achieved the 0.8 F1 score, and (ii) how many oracle decisions DExScore required to achieve this F1 score.

5.2.2 Results. We plot the fraction of runs that reached the desired threshold against the number of initial examples in Figure 7 and observe that for a relatively small number of examples (~20), all or nearly all of the runs will reach the desired threshold.

We also plot the number of oracle decisions versus the number of examples in Figure 8 and observe a tradeoff between the number of examples provided and the number of decisions required. For example, with an initial dataset of 20 labeled examples and 20 oracle decisions DExScore synthesizes programs that achieve the target F1 score. With an initial labeled set of 60 examples and 10 oracle decisions, the synthesized programs achieve the same F1 score. (We choose 20 labeled examples as a reference point here due to the results from the previous paragraph.) Intuitively, in this range, $23 - 12 = 11$ oracle decisions are as informative as $60 - 20 = 40$ labeled examples. This supports our hypothesis that providing oracle decisions about domain expertise allows synthesis to succeed with fewer examples.

5.3 Experiment 2: Answering RQ2 (on Ablating Predicate Suggestion)

5.3.1 Setup. We ran DExScore with our predicate suggestion algorithm (with 10, 20, and 30 starting examples) and compared it to a version of DExScore that randomly suggests predicates (this approach does not use any starting examples). Specifically, we ran these variants on the same task as Experiment 1, but instead of fixing an F1 threshold, we capped the number of oracle decisions for each run (ranging from 5 to 100 decisions) and tracked the mean F1 score across 50 runs for each decision count.

5.3.2 Results. We plot our results in Figure 9 and observe that our predicate suggestion algorithm outperforms the random strategy by a wide margin, with increasing gains as the number of initial examples increases. By 20 decisions, our approach achieves F1 scores above 0.75 on all example set sizes, whereas the random strategy achieves an F1 score of <0.1 and never achieves better than an F1 score of 0.3, even with 100 decisions.

5.4 Experiment 3: Answering RQ3 (on Comparing to an LLM Baseline)

5.4.1 Setup. For this experiment, we compare DExScore against an existing tool which can do a similar task. Since there are no tools targeted at this specific kind of task, we use an LLM, Gemini [Gemini Team (Google), 2024], as our point of comparison. Specifically, we use Gemini to decide whether an image does or does not depict a person playing sports (Sports Task), and whether a piece of music is similar to other pieces of music provided (Music Task). Gemini can accept both images and text.

As input for the Sports Task, we provide both the image and the set of features that DETR extracts for that image. Additionally, we provided the LLM with few-shot examples (including images, extracted features, and ground-truth labels), for a set of other images in the dataset, in the same way that we provide DExScore with examples, and for the same range of example set sizes. After varying the example set sizes, we report the highest F1 score the LLM achieved for any example set size, and compared it to the average F1 score of DExScore run with 20, 40, and 60 oracle decisions.

For the Music Task, we provide the chord progressions of the music piece to the LLM. Similarly to the Sports Task, we also provided the LLM with few-shot examples, in the form of other chord progressions. After varying the example set sizes, we report the highest F1 score the LLM achieved for any example set size, and compared it to the average F1 score of DExScore run with 20, 50, 100, and 150 oracle decisions.

As in RQ1, we assessed the F1 scores on the manually-labeled ground truth. To be as forgiving to the baseline as possible, we chose to display only the highest F1 score ever achieved by the LLM (as compared to the per-example-set averages for DExScore) because we observed that LLM performance decreased with more examples.

We note that the LLM approach is not a perfect baseline for our context. In particular, the LLM does not produce a program that can be read or modified, which is an explicit goal for our approach. Additionally, large language models are sensitive to the exact prompt that is used to obtain their output; it is possible that other prompts could result in better or worse performance.

5.4.2 Results. We show our results for this experiment on the Sports Task in Figure 10. We observed that for all numbers of oracle decisions tested (20, 40, 60), DExScore can outperform the LLM baseline. We observed a slight improvement when increasing from 20 to 40 oracle decisions, again showing the benefit of additional domain knowledge, but this boost appears to be quite small.

We show our results for this experiment on the Music Task in Figure 11. We observed that for all numbers of oracle decisions tested (20, 50, 100, 150), DExScore can match the LLM baseline given at least 5 examples. We observed no improvements from increasing the number of oracle decisions.

5.5 Experiment 4: Answering RQ4 (on Sensitivity to Extract and Ratio Variants)

In Section 4.3.1, we describe how our predicate suggestion algorithm depends on (i) an Extract function (to generate X^+ and X^- sets to compare with the log-likelihood ratio), and (ii) a Ratio function for doing the comparison. We described two variants of Extract: the direct and iterative strategy. We also described two variants of Ratio: with and without mutual information. For this component of the evaluation, we ran all four combinations of these variants to see if DExScore

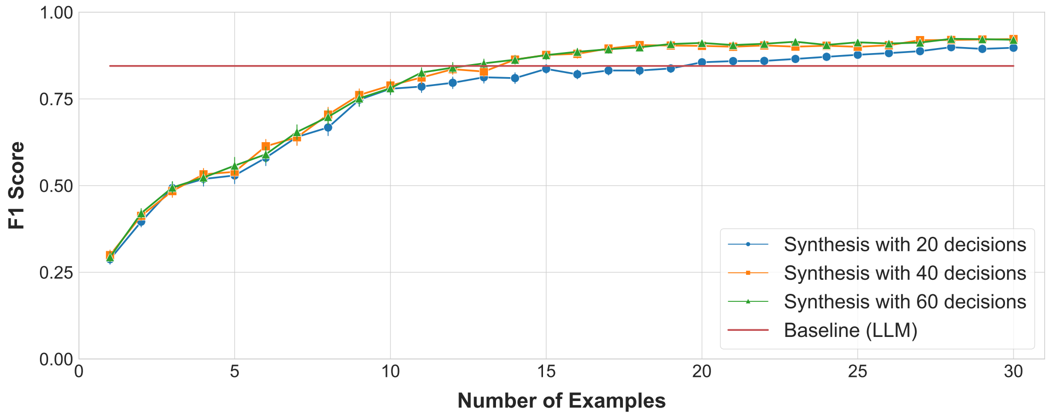

Fig. 10. **Synthesis performance on Sports Task as compared to an LLM baseline.** We gave an LLM a description of the Sports Task and asked it to label each document. It was given labeled examples as few-shot prompts. The baseline line represents the best performance across all numbers of few-shot examples given to the LLM. The x-axis represents the number of labeled input documents provided to DExScore, and the y-axis represents the F1 score computed against the hand-labeled ground truth (mean of 500 runs with 95% confidence interval error bars). The different lines represent different numbers of oracle decisions. On average, DExScore outperforms the LLM given at least 18 input examples and 20 oracle decisions.

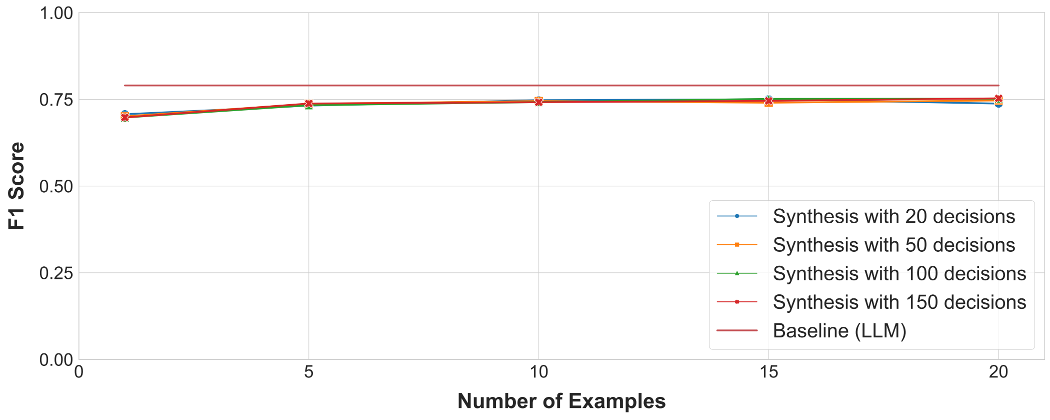

Fig. 11. **Synthesis performance on Music Task as compared to an LLM baseline.** We gave an LLM a description of the Music Task and asked it to label each document. It was given labeled examples as few-shot prompts. The baseline line represents the best performance across all numbers of few-shot examples given to the LLM. The x-axis represents the number of labeled input documents provided to DExScore, and the y-axis represents the F1 score computed against the hand-labeled ground truth (mean of 300 runs with 95% confidence interval error bars). The different lines represent different numbers of oracle decisions. On average, DExScore matches the LLM given at least 5 input examples and 20 oracle decisions.

was sensitive to the particular choices that went into our predicate suggestion algorithm. We ran these variants on the same task as Experiment 1, with a fixed number of oracle decisions (15). We tracked the mean F1 score of the variants over 50 runs for each number of examples.

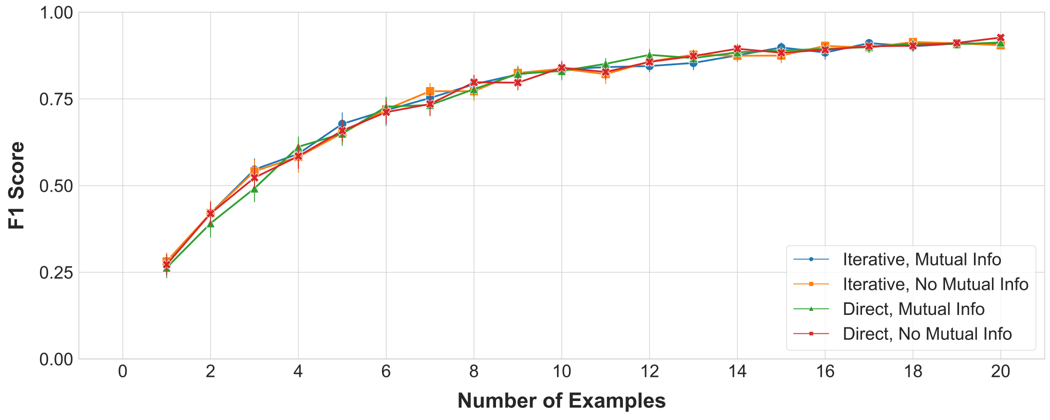

Fig. 12. **Synthesis performance with predicate suggestion algorithm variants.** We assessed DExScore's sensitivity to the details of the domain knowledge suggestion function. In particular, we ablated the Extract function (direct and iterative strategies), and the Ratio function (with and without mutual information filtering). Each line represents a different combination of these parameters. The x-axis represents the number of labeled input documents, and the y-axis represents the F1 score computed against the hand-labeled documents (mean of 50 runs with 95% confidence interval error bars). The *direct* strategy refers to the extractor function which simply sets X^+ equal to the initial set of user-provided input documents, and X^- to all other documents. The *iterative* strategy refers to the extractor function which also adds documents to X^+ and X^- using high and low thresholds on the current program.

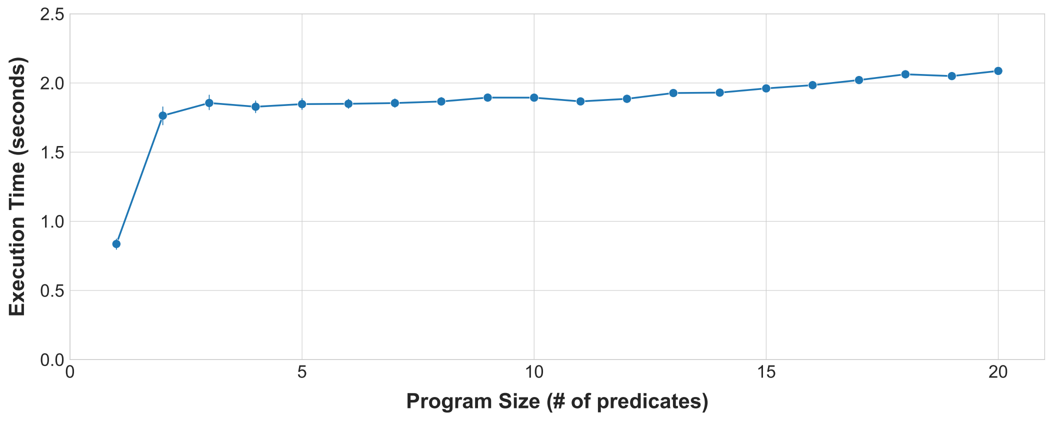

Fig. 13. **Execution time with increasing program sizes.** The x-axis is the size of the ground truth program, as measured by the number of predicates in it. The y-axis is the time it takes for DExScore to generate the ground-truth program given a randomly-generated ground truth program of a given size (mean of 500 random generations with 95% confidence interval error bars). We found that on average, it takes around 1.5-2.0 seconds for DExScore to synthesize the ground truth program across sizes from 2 to 20.

5.5.1 Results. We plot our results in Figure 12 and find that DExScore's performance is not sensitive to particular choices of Extract and Ratio functions.

5.6 Experiment 5: Answering RQ5 (on DExScore's Performance)

5.6.1 Setup. To investigate DExScore's scalability, we generated synthetic ground truth programs of increasing size and measured how long it took DExScore to synthesize these programs. For each program size from 1 to 20 (in terms of number of predicates), we randomly generated 500 programs and measured DExScore's execution time.

5.6.2 Results. We plot our results in Figure 13 and observe that for all program sizes larger than 1, the execution time is on average 1.5-2.0 seconds. The execution time increases slightly with increasing program size.

5.7 Threats to Validity

The particular hand-labeled task and randomly generated tasks we used to evaluate DExScore were not chosen based on their real-world utility for domain experts, but rather because they were (i) easily assessable using standard datasets and machine learning models, and (ii) a good testing ground to explore questions about whether domain knowledge can be explicitly incorporated and modeled into a synthesizer. Therefore, our results may not be representative of how DExScore would perform in real-world conditions.

6 DISCUSSION AND FUTURE WORK

As shown in Section 5.2, our approach enables trading labeling examples for making domain-relevant decisions in a synthesis loop. To answer this question, we used a programmatic oracle to make domain concept decisions, but an exciting avenue of future work would be to conduct a user study investigating how users navigate this tradeoff, and whether expressing their domain knowledge in this direct way can be less burdensome than labeling large numbers of examples, as in previous PBE approaches.

Future work could also explore new interfaces and interaction models for making these domain decisions. Currently, our interaction model enables an oracle to say that a predicate is or is not relevant with a yes/no response. In the future, we imagine richer predicates that have custom, modular user interfaces associated with them, in the style of Omar et al. [2021]'s livelits—for example, predicates that are parameterized by thresholds opening up slider interactions, or predicates to express that "at least *n* of the following must be present" opening up multi-select interactions.

Lastly, beyond the image classification instantiation explored in this work, we would be excited to apply this approach in other domains, particularly those where it is difficult or impossible for existing automated tools to have prior knowledge of the relevant domain knowledge. For example, in the music domain we briefly mentioned in Section 2, domain knowledge might actually be a user's own musical preferences about what musical features they like or find interesting. It would be interesting to see whether domain knowledge elicitation for this kind of task would be useful compared to approaches like a large language model (which will have only general knowledge, not user-specific knowledge) and programming-by-example (which cannot directly capture the domain knowledge).

7 RELATED WORK

The central research aim of this work is to evaluate the extent to which we can trade off examples with some other form of domain expertise intent specification. No other work explicitly evaluates this tradeoff, but a variety of approaches have been proposed that touch on some of the same areas that we do, which we survey below.

7.1 Synthesis Specifications

Complete specifications (like logical specifications and concrete programs) define precise specifications for a computational task, but may be hard for non-programmers to write. We therefore focus our discussion of related work on under-specifications (also known as partial specifications), adopting Le et al. [2017]'s practice of referring to PBE to denote "programming using under-specifications."

While requiring little background to use, input-output examples as a mode of specification suffer from a high degree of ambiguity and leave little control to the user [Lau, 2009, Mayer et al., 2015]. Consequently, a number of approaches have been taken to improve the PBE process. Drachsler-Cohen et al. [2017] propose a synthesis framework that relies on *abstract examples*, which are partial functions that describe the input-output behavior of the desired program—for example, to specify the multiplication function, the $x \times 1 = x$ abstract example represents the infinite set of concrete examples $\{1 \times 1 = 1, 2 \times 1 = 2, \ldots\}$. FlashSplit [Le et al., 2017] has a similar notion of "subset constraints" for their domain, which can be viewed as an abstract example in Drachsler-Cohen et al. [2017]'s framework. Similarly, An et al. [2019] use relational perturbation properties (e.g., specifying the length function `len` with `len([x, y, z]) = len([z, x, y])` to perform data augmentation [Dempster et al., 1977, Tanner and Wong, 1987] on a set of user-provided input-output examples.

Aside from abstract examples and properties, there has been recent work on supporting additional user annotations to describe more information to the synthesizer. In particular, Peleg et al. [2018b] and Peleg et al. [2020] introduce a system that allows users to annotate parts of candidate programs that should be kept in future iterations of synthesis, and Zhang et al. [2020] introduce a system that allows users to mark which parts of examples should be generalized, and to what extent. (Their system also supports data augmentation in the form of showing the user corner cases to disambiguate programs.)

Finally, as we mentioned in the introduction, there has been a long line of work on choosing which example to show to a user in programming-by-example systems [Jha et al., 2010, Padhi et al., 2018, Ji et al., 2020, Laich et al., 2020, Chen et al., 2023].

7.2 Iterative Program Synthesis

[Mayer et al., 2015], Le et al. [2017], Peleg et al. [2018a], Peleg et al. [2020], Hu et al. [2021], Zhang et al. [2021] introduce systems and frameworks to include users in the internal process of the synthesis algorithm. Such a user-in-the-loop process can also be considered an exploration of sketches (partial programs), as in program refinement [Dijkstra, 1968, Wirth, 1971], which some synthesizers explicitly explore [Li and Jagadish, 2014, Bastani et al., 2021, Ikarashi et al., 2021, Blinn et al., 2022]. One can also view programming-by-example tools that return partial programs [Chen and Weld, 2008] or programs that only satisfy parts of a specification [Peleg and Polikarpova, 2020] as part of such an iterative process.

7.3 Multi-Modal Program Synthesis

Multi-modal program synthesizers typically take in some of the kinds of specification mentioned above *and* natural language prompts [Chen et al., 2020, Ye et al., 2020, Rahmani et al., 2021, Ye et al., 2021]. These natural language prompts can be considered an elicitation of domain knowledge, but in an unstructured and uncheckable fashion; they have nonetheless proven useful in guiding synthesis algorithms. In our work, we model domain expertise explicitly and guarantee that returned programs respect the elicited domain knowledge.

7.4 Labeling Data

Our approach can be thought of as a way of labeling many images semi-automatically; existing approaches that do this fall into the categories of either weak supervisors or manual approaches.

Weak supervisors, popularized by Snorkel [Ratner et al., 2017], can be used to label data via rule-based classifiers, which need to be written by hand. Our work could be thought of as a way to produce a weak supervisor using program synthesis.

Alternatively, a variety of approaches have been proposed to assist or semi-automate human labeling of data [Cui et al., 2007, Russell et al., 2008, Tang et al., 2013, Andriluka et al., 2018, Dutta and Zisserman, 2019, Desmond et al., 2021a,b, Zhang et al., 2022], including some commercial options [Amazon Web Services, Inc., 2024, Refuel Team, 2024]. These approaches do not result in a program or any other artifact that can generalize to unseen examples.

Relatedly, semi-supervised learning can leverage small amounts of labeled data and large amounts of unlabeled data [Zhu and Ghahramani, 2002, Lee, 2013]; our system can thus be viewed as a kind of semi-supervised learning.

Finally, the line of work on example selection in programming-by-example can be considered a kind of active learning, in which systems query the user for data to label [Cohn et al., 1996, Freund et al., 1997, Roy and McCallum, 2001, Settles, 2009].

7.5 Synthesis of Machine Learning Combinators

Our approach synthesizes calls to black-box library functions (or metadata that happens to be available), and, in particular, DExScore synthesizes calls to feature extractions from a computer vision library. We do not require these functions to be machine learning algorithms, but there have been a variety of other neurosymbolic approaches that do rely on machine learning combinators more directly [Johnson et al., 2017, Barnaby et al., 2023, Li et al., 2023].

8 CONCLUSION

Using PBE in new domains can be difficult if it takes many examples to produce a sufficient specification. In answer to prior work suggesting that users may sometimes prefer to offer domain expertise rather that repetitive examples, we introduce a synthesis framework that accepts examples but also indications of whether particular domain concepts are relevant. Instantiating our framework in the domain of image classification allowed us to interrogate the tradeoff between providing examples versus domain knowledge in that space. We find that on average within our framework, an additional domain decision does indeed improve synthesized programs more than an additional labeled input-output example. This supports the intuition that domain expertise may be a more direct way of communicating intent than examples, in some cases. As far as we are aware, DExScore is the first synthesis framework that accepts both examples and domain concepts as inputs into a synthesis algorithm. We hope this initial framework opens the door to more synthesis approaches that integrate explicit domain expertise into specifications.

REFERENCES

Alfred V. Aho, Brian W. Kernighan, and Peter J. Weinberger. 1979. Awk — a Pattern Scanning and Processing Language. *Software: Practice and Experience* 9, 4 (1979), 267–279. https://doi.org/10.1002/spe.4380090403

Amazon Web Services, Inc. 2024. Machine Learning Labeling - Amazon SageMaker Ground Truth - AWS. http://web.archive.org/web/20240326155918/https://aws.amazon.com/sagemaker/groundtruth/.

Shengwei An, Rishabh Singh, Sasa Misailovic, and Roopsha Samanta. 2019. Augmented Example-Based Synthesis Using Relational Perturbation Properties. *Proceedings of the ACM on Programming Languages* 4, POPL (Dec. 2019), 56:1–56:24. https://doi.org/10.1145/3371124

Mykhaylo Andriluka, Jasper R. R. Uijlings, and Vittorio Ferrari. 2018. Fluid Annotation: A Human-Machine Collaboration Interface for Full Image Annotation. In *Proceedings of the 26th ACM International Conference on Multimedia (MM '18).*

Association for Computing Machinery, New York, NY, USA, 1957–1966. https://doi.org/10.1145/3240508.3241916

Celeste Barnaby, Qiaochu Chen, Roopsha Samanta, and Işıl Dillig. 2023. ImageEye: Batch Image Processing Using Program Synthesis. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 134:686–134:711. https://doi.org/10.1145/3591248

Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 2021. Synthesizing Queries via Interactive Sketching. https://doi.org/10.48550/arXiv.1912.12659 arXiv:1912.12659 [cs]

Andrew Blinn, David Moon, Eric Griffis, and Cyrus Omar. 2022. An Integrative Human-Centered Architecture for Interactive Programming Assistants. In *2022 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–5. https://doi.org/10.1109/VL/HCC53370.2022.9833110

Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. End-to-End Object Detection with Transformers. arXiv:2005.12872 [cs.CV]

Jiun-Hung Chen and Daniel S. Weld. 2008. Recovering from Errors during Programming by Demonstration. In *Proceedings of the 13th International Conference on Intelligent User Interfaces (IUI '08)*. Association for Computing Machinery, New York, NY, USA, 159–168. https://doi.org/10.1145/1378773.1378794

Qiaochu Chen, Xinyu Wang, Xi Ye, Greg Durrett, and Isil Dillig. 2020. Multi-Modal Synthesis of Regular Expressions. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. Association for Computing Machinery, New York, NY, USA, 487–502. https://doi.org/10.1145/3385412.3385988

Yanju Chen, Chenglong Wang, Xinyu Wang, Osbert Bastani, and Yu Feng. 2023. Fast and Reliable Program Synthesis via User Interaction. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 963–975. https://doi.org/10.1109/ASE56229.2023.00129

David A. Cohn, Zoubin Ghahramani, and Michael I. Jordan. 1996. Active Learning with Statistical Models. *Journal of Artificial Intelligence Research* 4, 1 (March 1996), 129–145.

Jingyu Cui, Fang Wen, Rong Xiao, Yuandong Tian, and Xiaoou Tang. 2007. EasyAlbum: An Interactive Photo Annotation System Based on Face Clustering and Re-Ranking. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems (CHI '07)*. Association for Computing Machinery, New York, NY, USA, 367–376. https://doi.org/10.1145/1240624.1240684

A. P. Dempster, N. M. Laird, and D. B. Rubin. 1977. Maximum Likelihood from Incomplete Data Via the EM Algorithm. *Journal of the Royal Statistical Society: Series B (Methodological)* 39, 1 (1977), 1–22. https://doi.org/10.1111/j.2517-6161.1977.tb01600.x

Michael Desmond, Evelyn Duesterwald, Kristina Brimijoin, Michelle Brachman, and Qian Pan. 2021a. Semi-Automated Data Labeling. In *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*. PMLR, 156–169.

Michael Desmond, Michael Muller, Zahra Ashktorab, Casey Dugan, Evelyn Duesterwald, Kristina Brimijoin, Catherine Finegan-Dollak, Michelle Brachman, Aabhas Sharma, Narendra Nath Joshi, and Qian Pan. 2021b. Increasing the Speed and Accuracy of Data Labeling Through an AI Assisted Interface. In *26th International Conference on Intelligent User Interfaces (IUI '21)*. Association for Computing Machinery, New York, NY, USA, 392–401. https://doi.org/10.1145/3397481.3450698

E. W. Dijkstra. 1968. A Constructive Approach to the Problem of Program Correctness. *BIT Numerical Mathematics* 8, 3 (Sept. 1968), 174–186. https://doi.org/10.1007/BF01933419

Dana Drachsler-Cohen, Sharon Shoham, and Eran Yahav. 2017. Synthesis with Abstract Examples. In *Computer Aided Verification*, Rupak Majumdar and Viktor Kunčak (Eds.). Springer International Publishing, Cham, 254–278. https://doi.org/10.1007/978-3-319-63387-9_13

Abhishek Dutta and Andrew Zisserman. 2019. The VIA Annotation Software for Images, Audio and Video. In *Proceedings of the 27th ACM International Conference on Multimedia (MM '19)*. Association for Computing Machinery, New York, NY, USA, 2276–2279. https://doi.org/10.1145/3343031.3350535

Yoav Freund, H. Sebastian Seung, Eli Shamir, and Naftali Tishby. 1997. Selective Sampling Using the Query by Committee Algorithm. *Machine Learning* 28, 2 (Aug. 1997), 133–168. https://doi.org/10.1023/A:1007330508534

Gemini Team (Google). 2024. Gemini: A Family of Highly Capable Multimodal Models. arXiv:2312.11805 [cs.CL]

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 '21)*. Association for Computing Machinery, New York, NY, USA, 134–148. https://doi.org/10.1145/3472749.3474740

Yuka Ikarashi, Jonathan Ragan-Kelley, Tsukasa Fukusato, Jun Kato, and Takeo Igarashi. 2021. Guided Optimization for Image Processing Pipelines. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576341

Susmit Jha, Sumit Gulwani, Sanjit A. Seshia, and Ashish Tiwari. 2010. Oracle-Guided Component-Based Program Synthesis. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 1. 215–224. https://doi.org/10.1145/1806799.1806833

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*

2020). Association for Computing Machinery, New York, NY, USA, 1143–1158. https://doi.org/10.1145/3385412.3386025

Justin Johnson, Bharath Hariharan, Laurens Van Der Maaten, Judy Hoffman, Li Fei-Fei, C. Lawrence Zitnick, and Ross Girshick. 2017. Inferring and Executing Programs for Visual Reasoning. In *2017 IEEE International Conference on Computer Vision (ICCV)*. 3008–3017. https://doi.org/10.1109/ICCV.2017.325

Larissa Laich, Pavol Bielik, and Martin T. Vechev. 2020. Guiding Program Synthesis by Learning to Generate Examples. In *International Conference on Learning Representations*.

Tessa Lau. 2009. Why Programming-By-Demonstration Systems Fail: Lessons Learned for Usable AI. *AI Magazine* 30, 4 (Oct. 2009), 65–65. https://doi.org/10.1609/aimag.v30i4.2262

Vu Le, Daniel Perelman, Oleksandr Polozov, Mohammad Raza, Abhishek Udupa, and Sumit Gulwani. 2017. Interactive Program Synthesis. https://doi.org/10.48550/arXiv.1703.03539 arXiv:1703.03539 [cs]

Dong-Hyun Lee. 2013. Pseudo-Label : The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks.

Fei Li and H. V. Jagadish. 2014. Constructing an Interactive Natural Language Interface for Relational Databases. *Proceedings of the VLDB Endowment* 8, 1 (Sept. 2014), 73–84. https://doi.org/10.14778/2735461.2735468

Ziyang Li, Jiani Huang, and Mayur Naik. 2023. Scallop: A Language for Neurosymbolic Programming. *Proceedings of the ACM on Programming Languages* 7, PLDI (June 2023), 166:1463–166:1487. https://doi.org/10.1145/3591280

Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. 2015. Microsoft COCO: Common Objects in Context. arXiv:1405.0312 [cs.CV]

Robert MacDougall. 1904. Recognition and Recall. *The Journal of Philosophy, Psychology and Scientific Methods* 1, 9 (1904), 229–233. https://doi.org/10.2307/2010991 jstor:2010991

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 '15)*. Association for Computing Machinery, New York, NY, USA, 291–301. https://doi.org/10.1145/2807442.2807459

Sydney Nguyen, Hannah McLean Babe, Yangtian Zi, Arjun Guha, Carolyn Jane Anderson, and Molly Q. Feldman. 2024. How Beginning Programmers and Code LLMs (Mis)Read Each Other. https://doi.org/10.48550/arXiv.2401.15232 arXiv:2401.15232 [cs]

Cyrus Omar, David Moon, Andrew Blinn, Ian Voysey, Nick Collins, and Ravi Chugh. 2021. Filling Typed Holes with Live GUIs. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 511–525. https://doi.org/10.1145/3453483.3454059

Saswat Padhi, Prateek Jain, Daniel Perelman, Oleksandr Polozov, Sumit Gulwani, and Todd Millstein. 2018. FlashProfile: A Framework for Synthesizing Data Profiles. *Proceedings of the ACM on Programming Languages* 2, OOPSLA (Oct. 2018), 150:1–150:28. https://doi.org/10.1145/3276520

Hila Peleg, Roi Gabay, Shachar Itzhaky, and Eran Yahav. 2020. Programming with a Read-Eval-Synth Loop. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (Nov. 2020), 159:1–159:30. https://doi.org/10.1145/3428227

Hila Peleg, Shachar Itzhaky, and Sharon Shoham. 2018a. Abstraction-Based Interaction Model for Synthesis. In *Verification, Model Checking, and Abstract Interpretation*, Isil Dillig and Jens Palsberg (Eds.). Springer International Publishing, Cham, 382–405. https://doi.org/10.1007/978-3-319-73721-8_18

Hila Peleg and Nadia Polikarpova. 2020. Perfect Is the Enemy of Good: Best-Effort Program Synthesis. In *DROPS-IDN/v2/Document/10.4230/LIPIcs.ECOOP.2020.2*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.ECOOP.2020.2

Hila Peleg, Sharon Shoham, and Eran Yahav. 2018b. Programming Not Only by Example. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. 1114–1124. https://doi.org/10.1145/3180155.3180189

Daniele Radicioni and Roberto Esposito. 2014. Bach Choral Harmony. UCI Machine Learning Repository. DOI: https://doi.org/10.24432/C5B61F.

Kia Rahmani, Mohammad Raza, Sumit Gulwani, Vu Le, Daniel Morris, Arjun Radhakrishna, Gustavo Soares, and Ashish Tiwari. 2021. Multi-Modal Program Inference: A Marriage of Pre-Trained Language Models and Component-Based Synthesis. *Proceedings of the ACM on Programming Languages* 5, OOPSLA (Oct. 2021), 158:1–158:29. https://doi.org/10.1145/3485535

Alexander Ratner, Stephen H. Bach, Henry Ehrenberg, Jason Fries, Sen Wu, and Christopher Ré. 2017. Snorkel: Rapid Training Data Creation with Weak Supervision. *Proceedings of the VLDB Endowment* 11, 3 (Nov. 2017), 269–282. https://doi.org/10.14778/3157794.3157797

Refuel Team. 2024. LLMs Can Label Data as Well as Humans, but 100x Faster. http://web.archive.org/web/20240204035221/https://www.refuel.ai/blog-posts/llm-labeling-technical-report.

N. Roy and A. McCallum. 2001. Toward Optimal Active Learning through Monte Carlo Estimation of Error Reduction. In *International Conference on Machine Learning*.

Bryan C. Russell, Antonio Torralba, Kevin P. Murphy, and William T. Freeman. 2008. LabelMe: A Database and Web-Based Tool for Image Annotation. *International Journal of Computer Vision* 77, 1 (May 2008), 157–173. https://doi.org/10.1007/s11263-007-0090-8

Burr Settles. 2009. *Active Learning Literature Survey.* Computer Sciences Technical Report 1648. University of Wisconsin–Madison.

C. E. Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27, 3 (July 1948), 379–423. https://doi.org/10.1002/j.1538-7305.1948.tb01338.x

Jinhui Tang, Qiang Chen, Meng Wang, Shuicheng Yan, Tat-Seng Chua, and Ramesh Jain. 2013. Towards Optimizing Human Labeling for Interactive Image Tagging. *ACM Transactions on Multimedia Computing, Communications, and Applications* 9, 4 (Aug. 2013), 29:1–29:18. https://doi.org/10.1145/2501643.2501651

Martin A. Tanner and Wing Hung Wong. 1987. The Calculation of Posterior Distributions by Data Augmentation. *J. Amer. Statist. Assoc.* 82, 398 (June 1987), 528–540. https://doi.org/10.1080/01621459.1987.10478458

Niklaus Wirth. 1971. Program Development by Stepwise Refinement. *Commun. ACM* 14, 4 (April 1971), 221–227. https://doi.org/10.1145/362575.362577

Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2021. Optimal Neural Program Synthesis from Multimodal Specifications. In *Findings of the Association for Computational Linguistics: EMNLP 2021*, Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (Eds.). Association for Computational Linguistics, Punta Cana, Dominican Republic, 1691–1704. https://doi.org/10.18653/v1/2021.findings-emnlp.146

Xi Ye, Qiaochu Chen, Xinyu Wang, Isil Dillig, and Greg Durrett. 2020. Sketch-Driven Regular Expression Generation from Natural Language and Examples. *Transactions of the Association for Computational Linguistics* 8 (2020), 679–694. https://doi.org/10.1162/tacl_a_00339

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 '21)*. Association for Computing Machinery, New York, NY, USA, 1–16. https://doi.org/10.1145/3411764.3445646

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 '20)*. Association for Computing Machinery, New York, NY, USA, 627–648. https://doi.org/10.1145/3379337.3415900

Yu Zhang, Yun Wang, Haidong Zhang, Bin Zhu, Siming Chen, and Dongmei Zhang. 2022. OneLabeler: A Flexible System for Building Data Labeling Tools. In *Proceedings of the 2022 CHI Conference on Human Factors in Computing Systems (CHI '22)*. Association for Computing Machinery, New York, NY, USA, 1–22. https://doi.org/10.1145/3491102.3517612

Xiaojin Zhu and Zoubin Ghahramani. 2002. Learning from Labeled and Unlabeled Data with Label Propagation.