

Exploring the Translation-As-Explanation Folk Wisdom for Program Editing Tasks

Jacob Yim^{*†}
UC San Diego
 La Jolla, USA
 jlyim@ucsd.edu

Justin Lubin^{*}
UC Berkeley
 Berkeley, USA
 jlubin@berkeley.edu

Kevin Ye[†]
Simon Fraser University
 Burnaby, CA
 yekeviny@sfu.ca

Laila Walker[†]
University of Washington
 Seattle, USA
 laila701@uw.edu

Eric Rawn
UC Berkeley
 Berkeley, USA
 erawn@berkeley.edu

Sarah E. Chasins
UC Berkeley
 Berkeley, USA
 schasins@cs.berkeley.edu

Abstract—Prior work in HCI and adjacent fields has often envisioned that a useful way to help programmers work with unfamiliar libraries or languages is to show a *translation*, an equivalent program written in a familiar library or language. However, prior work has not yet explored whether translations actually help programmers. Do they? Are there ways of presenting a given translation that make it more or less helpful? We conducted a 7-condition, 98-participant user study of program modification tasks, exploring whether a variety of translation-centric and non-translation interfaces helped participants. Surprisingly, there was a 67%–89% chance that translations alone made participants *less* likely to complete program modification tasks correctly. Our results lead us to question the intuition that translations are helpful for pragmatic purposes. More broadly, we highlight the need to better understand the practical knowledge required for program modification and adaptation—and how we can better design interventions that support these activities.

I. INTRODUCTION

Prior work in HCI and adjacent fields has speculated that autonomously-generated code can help programmers use unfamiliar libraries and languages [1]–[4], in particular by showing *translations* [5]–[9], which we define as equivalent programs written in a familiar language. However, this intervention has not yet been tested. We are not aware of any empirical evidence on whether translations help programmers work with unfamiliar programming abstractions.

Working with unfamiliar programs is difficult but ubiquitous; programmers must often read, edit, and reuse code written by other programmers and, increasingly, automatic code generators. With the recent popularity of programming with large language models (LLMs), there is a particularly urgent need to support programmers in working with code they did not write themselves. While we expect that programmers are familiar with the libraries and languages of their own code, that may not be the case for code written by peers or machines.

As a first exploration into the usefulness of translations, we study translation of unfamiliar library functions into a

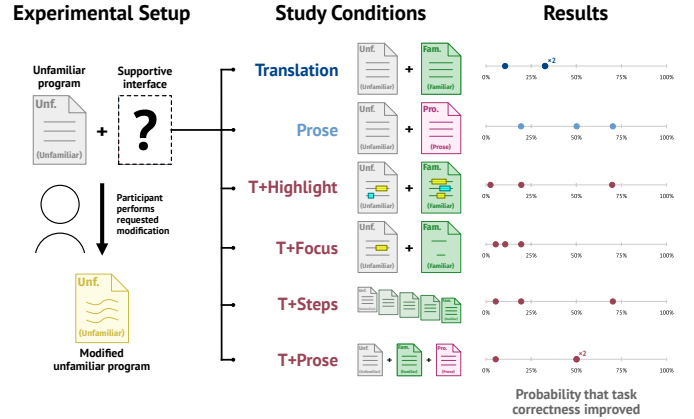


Fig. 1: In a controlled trial, we asked 98 Python programmers using translations and other supportive interfaces to modify unfamiliar TensorFlow programs and measured their correctness. Each participant saw one kind of interface: a **CONTROL** interface that showed only the unfamiliar program or one of six interfaces that offered additional information, such as a translation or prose explanation. Surprisingly, we found that participants in the **CONTROL** condition were the most correct.

familiar general-purpose host language. To investigate the folk wisdom that translations can help with modification tasks, we conducted a user study of 98 Python programmers editing programs with functions from a numerical computation library they were unfamiliar with, TensorFlow [10]. Our primary measurement of interest was correctness in code modification.

Our study revealed that Python translations alone did not help programmers work with TensorFlow. To probe further, we used an additional series of custom interfaces to investigate whether surfacing specific information about translations can make translations more helpful. These interfaces did make successful participants faster for a majority of tasks, but did not increase the number of successful participants.

^{*}Authors contributed equally.

[†]Work completed while at UC Berkeley.

Our results show that supporting programmers in modifying code that uses unfamiliar libraries is more complex than previously expected. In particular, the common intuition that translations help programmers work with unfamiliar languages does not explain our findings. We conclude with reflections on these findings to draw out implications for designing interfaces targeted at technical tasks like program modification.

Contributions. We present the following contributions:

- 1) **A 98-participant user study assessing participants’ success in editing programs using an unfamiliar library.** We compared between seven conditions, including five conditions that presented translations from unfamiliar TensorFlow functions into familiar Python.
- 2) **A set of exploratory interfaces designed to investigate different means of presenting program translations to users.** In addition to the translation-only interface, we evaluate four interfaces that augment translations with additional information. While some are novel in design, they are a secondary contribution in service to the study.

II. RELATED WORK

Despite speculation that translations may help programmers, there are no existing works that study users to identify whether translations help them with programming tasks. We therefore orient our discussion of prior studies of programmers into two nonoverlapping categories: studies that show program translations, and studies that ask programmers to work with programs they did not write themselves. We close with related work on analogical reasoning from domains outside of programming.

A. Studies Showing Programmers Translations

Shrestha et al.’s Transfer Tutor [5] is a tool that functions most similarly to what we investigate in our study. Transfer Tutor shows programmers a translation augmented with highlighted reusable concepts to promote the learning of a new programming language. From qualitative analysis of a think-aloud study of Transfer Tutor users, the authors concluded that users used *conceptual transfer* as a cognitive strategy. However, they did not assess whether translation affected programming task completion; there was no control condition that excluded code translation.

More broadly, the idea of using conceptual transfer to learn new programming languages has a rich history in the computer science education research community [11]–[20]. Providing translations is a common teaching method, with the aim of promoting conceptual transfer. As this research focuses on a classroom setting, we do not anticipate that its conclusions regarding the efficacy of translations apply to our study. In particular, we do not expect findings targeting learning outcomes under instruction to apply to short-term modification of unfamiliar code in a setting without an instructor, semester-long guided curriculum, or explicit learning goals.

B. Studies of Programmers Working with Code They Did Not Write

Since the literature includes few studies on the specific task that interests us—modifying code that includes unfamiliar

programming abstractions—we here widen our net to consider works that touch on possibly-related themes. For instance, it is possible that code literacy and comprehension may be a subtask within the overall code modification task. Our own work does not assess these topics, but we include an overview of the adjacent research landscape to orient readers.

1) *Working With Others’ Code Is Hard:* The literature offers evidence that working with code that one has not written oneself is difficult [21]–[26]. Studies reveal that the process is time-consuming [22], [23], [25], [26]. Especially relevant to our own context, a study of developers performing small code modification tasks uncovered that participants spent 35% of their time understanding unfamiliar code [23].

Moreover, LLMs are increasingly being used for code generation; while numerous studies have found that LLMs help experienced programmers write code [27]–[29], others have found that programmers struggle to use them effectively. Existing work highlights issues surrounding code comprehension and verification [30], [31] and difficulties engaging with unfamiliar abstractions in generated code [30]–[32]. We are interested in whether translations can address this need.

2) *Studies of Program Comprehension:* Since the 1970s, there has been a long line of work studying how novice and expert programmers engage with code written in familiar languages [21], [33]–[46]. A smaller line of inquiry has investigated how programmers comprehend code when working in unfamiliar languages [47], [48] or domains [49], including learning resources [7], [50]. Most relevantly, Shaft and Vessey [49] identify that program comprehension is easier if conducted in a familiar domain with an unfamiliar language compared to an unfamiliar domain with a familiar language. While these studies are relevant, we do not aim to study program comprehension; we are instead interested in the pragmatic ability to modify code. Moreover, none of these works examine translations.

A more specific line of work has investigated the particular *information needs* involved in program comprehension [23], [51]–[55], including some studies of programmers working with unfamiliar languages and APIs [52], [55] and some of developers performing pragmatic code modification tasks in a familiar language [23], [54]. These works present program understanding as being constructed from information needs specific to the task at hand. Most relevant to our study, Kirk et al. identified challenges faced by learners of an object-oriented framework, including during code modification activities [55].

3) *Studies of Program Explanations:* Regarding program explanations, prior work has proposed using pseudocode to explain unfamiliar programs [56]–[58]. Research has also proposed natural language explanations [59]–[61], with recent works emphasizing explanations generated by LLMs [62]. These works do not study effects on the pragmatic ability to modify code, nor do they study translations.

C. Analogical Reasoning

Program translations can be viewed as *analogies* that align unfamiliar code with familiar code based on shared input-

output behavior. There is a long history of work on analogical reasoning in cognitive science and HCI [63]–[68]. One foundational theory is Gentner’s structure-mapping theory [69], which characterizes analogy as a mapping of relations from a familiar base to an unfamiliar target. Within HCI, foundational work positions analogy as a powerful design tool for helping users understand unfamiliar user interfaces [66], [67], but also cautions that poorly constrained analogies can mislead users [64]. These prior works on analogical thinking motivate our augmented interface designs (Section IV-C) and help situate and explain our findings (Section VII-A).

III. RESEARCH QUESTIONS

We explore the following research questions:

- RQ1** Do translations help programmers work with unfamiliar abstractions?
- RQ2** How does translation compare to an alternative intervention—natural language explanation—in helping programmers work with unfamiliar abstractions?
- RQ3** Can translations be augmented to be more helpful to programmers working with unfamiliar abstractions?

To answer **RQ1**, we compare programmer performance on a program editing task when given two kinds of information:

- 1) The program using unfamiliar abstractions (starter code).
- 2) The program using unfamiliar abstractions (starter code) and a translation using familiar abstractions.

To answer **RQ2**, we compare programmer performance on a program editing task when given:

- 3) The program using unfamiliar abstractions (starter code) and a natural-language explanation (with *no* translation).

To answer **RQ3**, we started with a set of four hypotheses about information that might make translations more helpful to programmers. **RQ3** explores these hypotheses in particular:

- H1** A mapping between familiar and unfamiliar program components helps.
- H2** Translation of individual, focused program components in isolation helps.
- H3** Seeing gradual, fine-grained translation steps helps.
- H4** Natural-language explanation of the translation helps.

To explore these hypotheses, we designed four interfaces augmenting translations in specific ways (Section IV-C).

IV. INTERFACES

In this section, we discuss the design and implementation of the programming interfaces associated with each condition in our study. In addition to the RQ-relevant features highlighted below, all interfaces featured the ability to read, edit, and run Python programs.

A. Control and Translation Interfaces

We used two interfaces to assess **RQ1**, **CONTROL** and **TRANSLATION**. The **CONTROL** interface displays only the original TensorFlow program (Figure 2a). The **TRANSLATION** interface displays the original TensorFlow program and a translation to vanilla Python code (Figure 2b). To faithfully

address the speculation around the usefulness of *automated* translation tools, we produced all translations for this study using an automated transpiler [70]. In particular, the transpiler automatically rewrites vectorized TensorFlow functions into `for` loops in base Python. The transpiler recognizes a subset of TensorFlow library functions, and incorporates compiler techniques, including inlining and a set of rewrite rules, to produce translations.

B. Natural Language Interface

The interface we used to assess **RQ2**, **PROSE** (Figure 2c), displays the original TensorFlow program alongside a natural language explanation. We produced explanations using OpenAI’s GPT-4 language model [71], the most recent version of the GPT model at the time this research was conducted in 2024. The model was prompted with the following text: “Explain this TensorFlow program concisely:”, followed by the TensorFlow starter code. We generated one explanation per task in advance; all participants in this condition saw the same text.

C. Augmented Interfaces

Lastly, we developed four augmented interfaces to assess hypotheses **H1–H4** of **RQ3**.

These interfaces used the same translations as **TRANSLATION**, with additional interface elements generated by extracting translation-related information from the transpiler. We intentionally constrained our augmented interfaces to information that can be collected automatically from a translation tool so that we can expect these interfaces to be automatically generatable rather than relying on human guidance or insights.

1) *Augmented Interface 1 (T+HIGHLIGHT): Highlighting Correspondence Between Translated Components:* We designed this interface to explore hypothesis **H1: A mapping between familiar and unfamiliar program components helps**.

T+HIGHLIGHT displays the source program, the translation, and a set of buttons. When clicked, each button highlights one of the TensorFlow functions in the source program and the corresponding code in the translation. It also highlights each function argument in its own color, in both the source and translated programs.

For example, Figure 2d shows **T+HIGHLIGHT** after clicking the `tf.math.reduce_sum` button. This function computes the sum of the elements in a tensor. The function call in the original program and corresponding translated code are highlighted. The argument of the function, `tf.math.multiply(x, 2)`, highlighted in red, determines both the number of iterations of the translated `for` loop and the value added to `sum_result` at each iteration. Thus, these parts of the translation are also highlighted in red.

This design lets us explore whether making the mapping between components salient may play a role in programmers’ translation use. From an analogical reasoning perspective, **T+HIGHLIGHT** operationalizes structure-mapping theory [69] by making the mapping of relations between the unfamiliar

For reference, here is the original TensorFlow program:

```
new_x = tf.math.reduce_sum(tf.math.multiply(x, 2))
```

(a) **CONTROL (RQ1)**: shows just the starter code.

For reference, here is the original TensorFlow program:

```
new_x = tf.math.reduce_sum(tf.math.multiply(x, 2))
```

Here is a translation of this program into a Python for loop:

```
sum_result = 0
for sum_i in range(len(x)):
    sum_result = sum_result + (x[sum_i] * 2)
new_x = sum_result
```

(b) **TRANSLATION (RQ1)**: shows the starter code and a translation.

For reference, here is the original TensorFlow program:

```
new_x = tf.math.reduce_sum(tf.math.multiply(x, 2))
```

Here is an explanation of this program:

This TensorFlow program performs two main operations on a tensor x:

1. **Element-wise Multiplication**: `tf.math.multiply(x, 2)` multiplies each element in the tensor x by 2.
2. **Sum Reduction**: `tf.math.reduce_sum(...)` computes the sum of all the elements in the resulting tensor after the multiplication.

In essence, the program doubles each element in the tensor x and then calculates the sum of all these doubled values.

(c) **PROSE (RQ2)**: shows the starter code and a natural language explanation.

Below is a Python translation of the original TensorFlow program. Click the buttons to highlight the parts of the Python program that correspond to the selected component of the TensorFlow program.

`tf.math.multiply` `tf.math.reduce_sum` `clear`

```
new_x = tf.math.reduce_sum(tf.math.multiply(x, 2))
sum_result = 0
for sum_i in range(len(x)):
    sum_result = sum_result + (x[sum_i] * 2)
new_x = sum_result
```

(d) **T+HIGHLIGHT (RQ3)**: shows the starter code and a translation, highlighting corresponding components.

Click the buttons below to show translations of individual parts of the program.

`tf.math.reduce_sum` `tf.math.multiply`

Here is the original TensorFlow program:

```
new_x = tf.math.reduce_sum(tf.math.multiply(x, 2))
```

Here is the translated component:

```
sum_result = 0
for sum_i in range(len(x)):
    sum_result = sum_result + (x[sum_i] * 2)
new_x = sum_result
```

(e) **T+FOCUS (RQ3)**: shows the starter code and buttons that translate individual components.

Press the "Translate one step" button to show the steps translating the TensorFlow code into Python. Differences from the previous step are highlighted.

```
tf.math.reduce_sum(tf.math.multiply(x, 2))
sum_result = 0
for sum_i in range(len(x)):
    sum_result = sum_result + tf.math.multiply(x, 2)[sum_i]
new_x = sum_result
sum_result = 0
for sum_i in range(len(x)):
    sum_result = sum_result + tf.math.multiply(x, 2)[sum_i]
new_x = sum_result
sum_result = 0
for sum_i in range(len(x)):
    sum_result = sum_result + (x[sum_i] * 2)
new_x = sum_result
Go back one step
```

(f) **T+STEPS (RQ3)**: shows the starter code, a translation, and a sequence of intermediate translation steps.

For reference, here is the original TensorFlow program:

```
new_x = tf.math.reduce_sum(tf.math.multiply(x, 2))
```

Here is a translation of this program into a Python for loop:

```
sum_result = 0
for sum_i in range(len(x)):
    sum_result = sum_result + (x[sum_i] * 2)
new_x = sum_result
```

Here is an explanation of these programs:

These TensorFlow and Python programs are equivalent because they both double each element of the tensor x and then sum the results:

- **TensorFlow**: `tf.math.multiply(x, 2)` doubles each element of x, and `tf.math.reduce_sum(...)` sums these doubled values.
- **Python**: The loop multiplies each element of x by 2 and accumulates the sum in `sum_result`.

Both methods produce the same final sum.

(g) **T+PROSE (RQ3)**: shows the starter code, a translation, and a natural language explanation of why the translation is equivalent.

Fig. 2: **The seven interfaces we use to answer our research questions.** Each interface specifically helps answer one of our research questions from Section III. **RQ1** asks whether translations help programmers work with unfamiliar abstractions; we compare **CONTROL** (a) and **TRANSLATION** (b) to answer it. **RQ2** asks whether natural language explanations help; we additionally compare **PROSE** (c) to answer it. **RQ3** asks whether translations can be augmented to be more helpful; we compare the augmented interfaces (d–g) to answer it.

target (TensorFlow) and the familiar base (Python) explicit. In the Cognitive Dimensions of Notations framework [72], this approach may improve *role-expressiveness* by mapping components in the original program to their behavior.

2) **Augmented Interface 2 (T+FOCUS): Translating Individual Components**: We designed this interface to explore hypothesis **H2: Translation of individual, focused program components in isolation helps**.

T+FOCUS does not display a whole-program translation of the original TensorFlow program. Instead, it provides a set of buttons that show translations of individual components of the original TensorFlow program. Each button corresponds to a TensorFlow function call in the original program. When these buttons are clicked, **T+FOCUS** underlines the corresponding function call in the original program and displays a translation of the underlined component. **T+FOCUS** provides a translation of the entire program only when the user clicks the button that corresponds to the outermost TensorFlow function call.

For example, Figure 2e shows **T+FOCUS** after clicking the button for `tf.math.multiply`. **T+FOCUS** underlines this function call in the source program and displays only this function’s translation.

The design of **T+FOCUS**, like **T+HIGHLIGHT**, leverages structure-mapping theory and role-expressiveness by mak-

ing relations between translation inputs and outputs explicit. However, unlike **T+HIGHLIGHT**, **T+FOCUS** allows users to interactively select what to translate, effectively adjusting the *abstraction level* [72] of what is displayed. This design was loosely inspired by the “step in” and “step out” feature of modern debuggers, which help programmers reveal behavior or abstract away unnecessary details by managing the abstraction level of visible information.

3) **Augmented Interface 3 (T+STEPS): Step-by-Step Translations**: We designed this interface to explore hypothesis **H3: Seeing fine-grained translation steps helps**.

T+STEPS initially shows only the original TensorFlow program and a “Translate one step” button. When the user clicks the step button, **T+STEPS** displays one step of the translation process, highlighting differences from the previous step. The user can click the step button repeatedly until the program is fully translated. **T+STEPS** also displays a “Go back one step” button to hide the latest step.

For example, Figure 2f shows **T+STEPS** after clicking the “Translate one step” button repeatedly until the program is fully translated. The final block of code is a full translation, and all intermediate blocks are partial translations en route to the final translation.

This design was inspired by “stepping” in debuggers, in

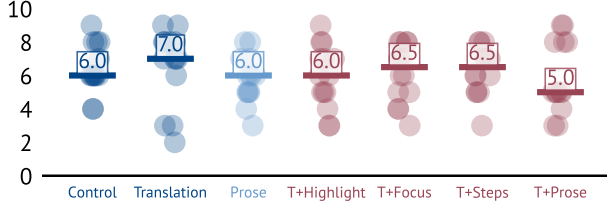


Fig. 3: Participants’ self-reported experience per condition.

that displaying information incrementally may help users understand the effects of each step in a larger process.

4) *Augmented Interface 4 (T+PROSE): Translations Supported by Natural Language Explanations*: We designed this interface to explore hypothesis **H4: Natural-language explanation of the translation helps**.

We posit that an explanation of the translation written in natural language may make translations more useful to programmers. **T+PROSE** thus shows the original TensorFlow program, a translation, and a natural language explanation of why the translation is equivalent. We produced natural language explanations using the same LLM as **PROSE**. We prompted the model using the text: “Concisely explain why the following TensorFlow and Python programs are equivalent:”, followed by the original TensorFlow program, then the Python translation. As with **PROSE**, we ran all prompts ahead of time so that all participants saw the same explanation.

V. STUDY DESIGN

To answer our research questions, we selected Python as the familiar language of study, and TensorFlow as the unfamiliar library. We selected Python, a popular general-purpose programming language, to improve generalizability to other popular languages. We selected TensorFlow as a popular library with enough complexity to present challenges for unfamiliar users. This approach also enhances ecological validity because programmers may frequently encounter unfamiliar libraries within a familiar host language, especially in LLM-generated or peer-written code. We opted to evaluate relatively small-scale translations—from unfamiliar library functions into a familiar host language—as a first investigation into the utility of translations.

We conducted a 7-condition 98-participant between-subjects study. The primary parameter we are interested in estimating is the probability that a participant using a particular interface succeeds on a particular task. As a secondary metric, we measured the time each participant took using an interface on a task. Because we are interested in studying participants’ abilities to pragmatically modify and adapt unfamiliar code rather than program comprehension, learning outcomes, or other cognitive phenomena, we restricted our attention to these metrics and did not administer, for example, validated concept inventories (to measure comprehension) nor validated workload indexes (such as the NASA TLX, to measure perceived

load). Our full dataset and analysis scripts are freely available online at <https://github.com/jcbym/translations-vlhcc26>.

A. Study Structure

a) *Participants and Recruiting*: We screened participants using a survey asking about prior experience with Python and a variety of Python libraries (including TensorFlow). The screening survey also included a quiz asking respondents to predict the output of three snippets of Python code. We recruited participants who (i) self-identified as having experience with Python, (ii) did not self-identify as having experience with TensorFlow, and (iii) correctly predicted the output of all three Python snippets. We recruited 98 participants (14 per condition), primarily through university-affiliated mailing lists, newsletters, and forums, and snowball sampling. We conducted one-on-one study sessions via Zoom, compensating each participant with a 30 USD Amazon gift card. This study was approved by our institution’s Institutional Review Board.

We balanced self-identified programming experience levels across the conditions using a survey question proposed and validated by Feigenspan et al. [73]. Figure 3 shows the distribution of self-identified programming experience by condition. Table I shows that the secondary metrics of years of experience and familiarity with other libraries are roughly balanced among conditions.

b) *Study Protocol*: We conducted study sessions via a webpage with a simple editing interface, as shown in Figure 4. We chose to use a lightweight editor instead of a traditional IDE due to low task complexity and to eliminate the need for additional software. We asked each participant to complete a tutorial task, then three program editing tasks (Section V-B). Each participant used only a single interface. We ran a between-subjects study in which participants completed tasks in a fixed order in one of seven conditions, corresponding to the interfaces described in Section IV. All interfaces featured the ability to read, edit, and run Python programs, as well as to skip the task entirely.

After informed consent, we guided participants through the tutorial. We explained the study format and the features of the study interface, including elements specific to the participant’s assigned condition. At the end of the tutorial, participants completed the tutorial task, a simpler modification task than the main study tasks. After completing this task, participants proceeded to the main study tasks.

During the main study tasks, participants were permitted to use all internet resources and software tools except those capable of code generation (e.g. ChatGPT). Internet access was permitted to improve external validity, as online resources may be useful to real-world programmers working with unfamiliar libraries. However, code generation resources were restricted to limit access to direct answers to tasks. Study sessions were capped at 90 minutes, after which participants “timed out” and the session was ended. Text written in the code editor and interactions on the webpage were periodically logged by the study interface. Following the study, we determined participant

	YoE	WEEKLY YoE	beautiful_soup	flask	matplotlib	nltk	numpy	pandas	pytorch
CONTROL	4.0	2.5	3	0	12	2	14	11	1
TRANSLATION	3.5	2.5	3	0	9	0	13	10	3
PROSE	4.0	2.0	3	3	10	0	12	10	2
T+HIGHLIGHT	5.5	2.0	4	3	8	2	14	10	4
T+FOCUS	4.0	3.0	3	5	9	1	12	11	3
T+STEPS	4.5	4.0	4	2	11	0	11	12	3
T+PROSE	5.0	2.0	2	3	9	0	11	10	2

TABLE I: **The number of participants in each condition that had worked with a variety of existing libraries.** All participants indicated they had **not** worked with the library we used in this study, TensorFlow. The three most similar libraries, numpy, pandas, and pytorch, had roughly the same number of participants that had worked with them in each condition.

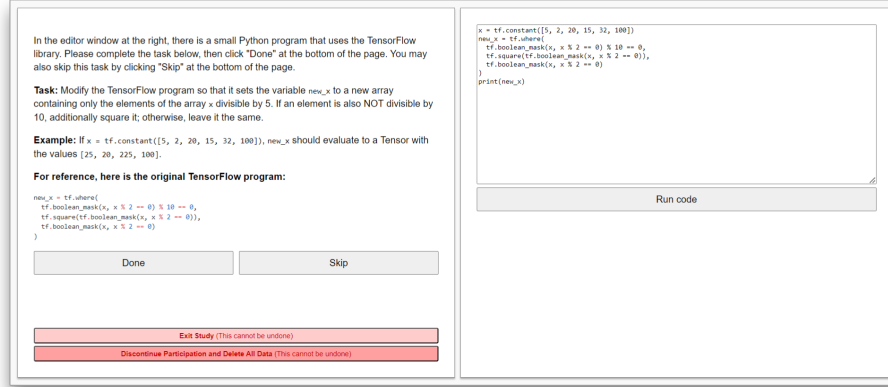


Fig. 4: **The main part of the web interface for the study.** Participants used this interface (extended in a variety of ways we describe in Sections IV-A, IV-B, and IV-C and show in Figure 2) to complete program modification tasks. On the left is a (i) description of the interface, (ii) a program modification task, (iii) the starter code, and (iv) buttons to submit the current code, skip the task, exit the study, and discontinue participation and delete all data. On the right is (i) a code editor initialized with the starter code and (ii) a run button that runs the current code and displays the result.

correctness using a series of property-based tests, which were not shown to participants.

B. Tasks

The three program modification tasks used during our study, alongside a tutorial task, are shown in Table II. Each task included starter code incorporating functions from the TensorFlow library and a new expected behavior, as well as an input-output example of the expected behavior. The tutorial task was used to introduce participants to the study format, and no data from this task was recorded. Tasks primarily involved array manipulation using TensorFlow’s vector operations; participants demonstrated basic prior knowledge of arrays during the screening survey.

C. Statistical Model

We use Bayesian inference to quantify the uncertainty in our measurements in light of best practices for fair statistical communication in HCI [74], [75]. For our primary outcome of interest, correctness, we define:

- i_n : the interface that participant n used (**observed** categorical variable with 7 categories).
- $c_{n,\tau}$: whether or not participant n successfully completed task τ (**observed** binary variable).

- $\theta_{i,\tau}$: the probability that a participant using interface i succeeds on task τ (**latent** variable taking on values between 0 and 1). We use a uniform prior on $\theta_{i,\tau}$ ranging from 0 to 1.

Intuitively, whether or not a participant using an interface i succeeded on task t is like a weighted coin flip with success probability $\theta_{i,t}$. We therefore use a Bernoulli distribution for our model:

$$c_{n,\tau} \sim \text{Bernoulli}(\theta_{i_n,\tau}).$$

For our secondary measure, time taken (in minutes), we define:

- $t_{n,\tau}$: the time participant n took on task τ (**observed** positive real variable).
- $\mu_{c,i,\tau}$: the mean log time taken a participant with correctness c (i.e., incorrect or correct) using interface i spends on task τ (**latent** real variable). We use a uniform prior on $\mu_{c,i,\tau}$ ranging from 0 to $\log(90)$, the duration of the study.
- $\sigma_{c,\tau}$: the (pooled) standard deviation of the log time taken of participants with correctness c on task τ (**latent** real variable). We use a uniform prior on $\sigma_{c,\tau}$ ranging from 0 to $\log(90)$, the duration of the study.

TABLE II: **Program modification tasks for our study.** Data from the tutorial task was not included in any analysis.

NAME	TASK DESCRIPTION	STARTER CODE
Tutorial	Modify the TensorFlow program so that it sets the variable <code>new_x</code> to the sum of the product of each entry of <code>x</code> with 3. Example: If <code>x = tf.constant([1, 1, 2, 3, 5])</code> , <code>new_x</code> should evaluate to a Tensor with the value 36.	<pre>new_x = tf.math.reduce_sum(tf.math.multiply(x, 2))</pre>
Task 1	Modify the TensorFlow program so that it sets the variable <code>new_x</code> to a new array containing only the elements of the array <code>x</code> divisible by 5. If an element is also NOT divisible by 10, additionally square it; otherwise, leave it the same. Example: If <code>x = tf.constant([5, 2, 20, 15, 32, 100])</code> , <code>new_x</code> should evaluate to a Tensor with the values [25, 20, 225, 100].	<pre>new_x = tf.where(tf.boolean_mask(x, x % 2 == 0) % 10 == 0, tf.square(tf.boolean_mask(x, x % 2 == 0)), tf.boolean_mask(x, x % 2 == 0))</pre>
Task 2	Modify the TensorFlow program so that it sets the variable <code>new_x</code> to a rolling weighted sum of array <code>x</code> with a window size of 2. The weights should be 2 and 1. That is, the first element of the new array should be the first element of <code>x</code> multiplied by 2 plus the second element of <code>x</code> multiplied by 1; the second element of the new array should be the second element of <code>x</code> multiplied by 2 plus the third element of <code>x</code> multiplied by 1; and so on. Example: If <code>x = tf.constant([4, 12, 8, 8])</code> , <code>new_x</code> should evaluate to a Tensor with the values [20, 32, 24].	<pre>new_x = tf.squeeze(tf.nn.conv1d(tf.reshape(x, [1, int(x.shape[0]), 1]), tf.constant([[[[1]], [[1]], [[1]]]], 1, 'VALID')))</pre>
Task 3	Modify the TensorFlow program so that it sets the variable <code>new_x</code> to the maximum of the minimum of every other sub-array in the array <code>x</code> (that is, the array at index 0, index 2, index 4, etc.). Example: If <code>x = tf.constant([[1, 2, 3], [101, 102, 103], [11, 12, 13]])</code> , <code>new_x</code> should evaluate to a Tensor with the value 11.	<pre>new_x = tf.math.reduce_min(tf.math.reduce_max(tf.boolean_mask(x, tf.range(len(x)) % 3 == 0), axis=1))</pre>

As time taken is positive and right-skewed, we model it with a log-normal model:

$$\log(t_{n,\tau}) \sim \text{Normal}(\mu_{c_n, i_n, \tau}, \sigma_{c_n, \tau}^2).$$

VI. RESULTS

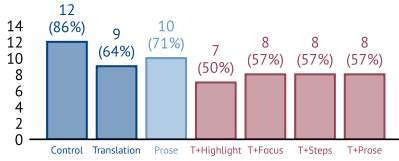
Figure 5 shows our raw data for correctness, and Figure 6 shows our raw data for time taken among correct participants. In addition, two of 98 participants, one in the **TRANSLATION** condition and one in the **T+STEPS** condition, “timed out” by reaching the 90-minute cap on session duration; both answered incorrectly on the task during which they timed out, so neither contributes to our time-taken analysis. Using Stan [76], we calculated the posterior distribution given our observed data of the effect sizes of the parameters of the statistical models we defined in Section V-C relative to the **CONTROL** condition. Figure 7 shows the effect sizes for probability of success (difference in probabilities) and Figure 8 shows the effect sizes for time taken (difference in median time taken) among correct participants. Each posterior plot also indicates the posterior probability that the interface is better than the control for the given parameter; these probabilities are what we used to create Figure 1, and, in particular, are not frequentist p -values.

1) *RQ1: Translations Alone Likely Do Not Help Programmers Work With Unfamiliar Abstractions:* To answer **RQ1**, we compare **CONTROL** and **TRANSLATION**. Figure 7 shows that **TRANSLATION** has an 11%, 33%, and 32% chance of having a higher probability of success than **CONTROL** for Tasks 1,

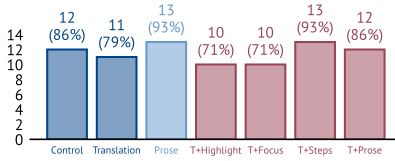
2, and 3 respectively. Among correct participants, Figure 8 shows that **TRANSLATION** has a 49%, 50%, and 1% chance of having a faster time taken than **CONTROL** for Tasks 1, 2, and 3 respectively. Participants using **TRANSLATION** did not perform better than **CONTROL** on either measure for any task.

2) *RQ2: Natural Language Explanations Moderately Speed Work With Unfamiliar Abstractions For Correct Participants While Maintaining Roughly Comparable Correctness:* To answer **RQ2**, we compare **CONTROL** and **PROSE**. Figure 7 shows that **PROSE** has a 19%, 70%, and 50% chance of having a higher probability of success than **CONTROL** for Tasks 1, 2, and 3 respectively. Among correct participants, Figure 8 shows that **PROSE** has a 99%, 97%, and 40% chance of having a faster time taken than **CONTROL** for Tasks 1, 2, and 3 respectively. The most credible estimate for the effect sizes for **PROSE** are 6.6 and 5.4 minutes faster than **CONTROL** for Task 1 and 2 respectively. For Task 3, the most credible estimate is 0.2 minutes slower. Overall, **PROSE** and **CONTROL** each have a task they perform better on than the other on correctness, and **PROSE** performs better than **CONTROL** on time taken for 2/3 tasks.

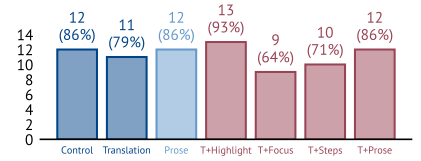
3) *RQ3: Augmented Translations Moderately Speed Work With Unfamiliar Abstractions For Correct Participants:* To answer **RQ3**, we compare **CONTROL** to our four augmented interfaces (**T+HIGHLIGHT**, **T+FOCUS**, **T+STEPS**, **T+PROSE**). Figure 7 shows that the augmented interfaces follow roughly the same trend as **TRANSLATION** for correctness, with two main exceptions: **T+STEPS** has a 70% chance of having



(a) Task 1 correct responses.

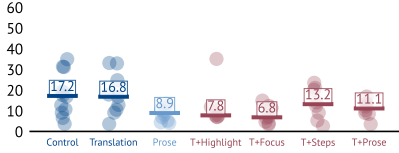


(b) Task 2 correct responses.

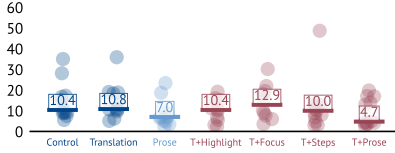


(c) Task 3 correct responses.

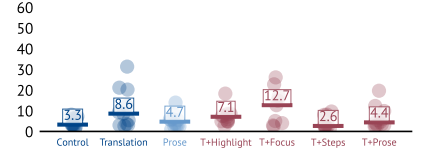
Fig. 5: Number of correct responses for each task and interface (raw data).



(a) Task 1 time taken (correct participants).



(b) Task 2 time taken (correct participants).



(c) Task 3 time taken (correct participants).

Fig. 6: Time taken among correct participants for each task and interface (raw data).

a higher probability of success than **CONTROL** on Task 2, and **T+HIGHLIGHT** has a 70% chance of having a higher probability of success on Task 3. In the first case, the most credible estimate for the effect size is a 5% difference in probability of success; for the second, 7%. Otherwise, no augmented interface is more likely to have a higher probability of success than **CONTROL**. On the other hand, Figure 8 shows that for Tasks 1 and 2, the augmented interfaces were always more likely than not to decrease time taken, with effect sizes between 0.6 minutes and 7 minutes faster. For Task 3, only **T+STEPS** was more likely to decrease time taken, with the most credible estimate for effect size being 0.6 minutes faster.

A. Threats to Validity and Limitations

a) *Threats to Validity*: Comparing time taken only among successful participants may introduce survivorship bias. An alternative study design could ask participants to keep re-submitting their answer until it was correct. This design would assume participants have access to a correctness oracle, which may reduce ecological validity.

Participants used a minimal editing interface alongside specialized per-condition interfaces. The absence of other interface features may have pushed participants to focus artificially on the optional aids. Participants completed tasks in a fixed order, so our study design did not control for potential confounding factors including warm-up time and learning effects. Participants may have also experienced time pressure, as sessions were capped at 90 minutes. Additionally, LLM outputs for the **PROSE** and **T+PROSE** conditions were fixed across participants, so our findings may not generalize across other natural-language or LLM-generated explanations.

The interfaces, translations, and LLM prompts used in this study are not representative of all possibilities that test our research questions; it is possible that other designs would yield different outcomes. Moreover, our interfaces and others may

perform differently on different tasks or task domains, such as comprehension of large codebases.

Our study involved translations of functions from a library (TensorFlow) into a general-purpose host language (Python). Our findings may not generalize to other types of translations (*e.g.*, from one general-purpose language to another). Additionally, we used one particular library and language for our study, and our results may not generalize to choices substantially different from these.

Finally, our findings may not generalize to populations beyond the one we studied, which consisted of programmers already familiar with Python (but not TensorFlow) who primarily were students and recent alumni of R1 universities in the United States.

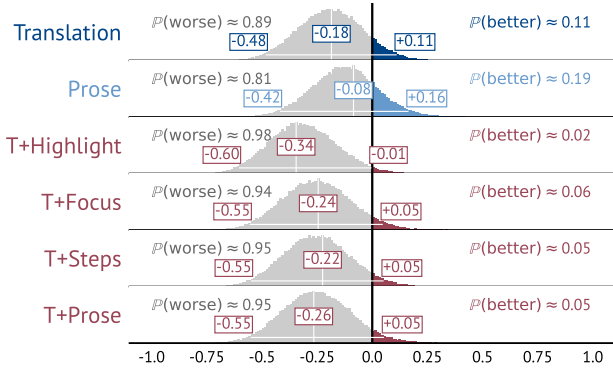
b) *Limitations*: This study is limited in that we did not conduct qualitative data analysis, which could provide nuanced insights into participants' experience of each interface. Lastly, we emphasize that, although it may be tempting to use our data about program modification to draw conclusions about program comprehension or learning outcomes, the ability to work with and modify code does not necessarily require a deeper understanding of the programs.

VII. DISCUSSION

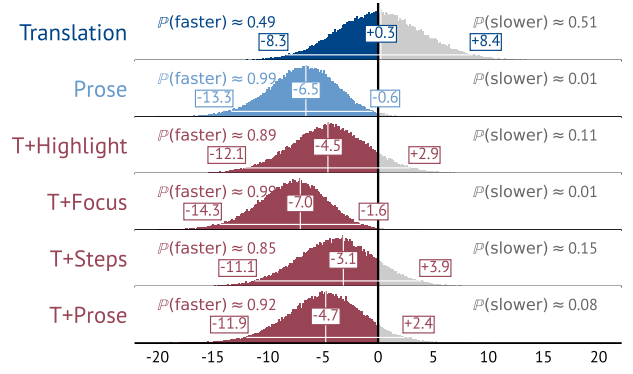
A. Interpretation of Results

In the following section, we offer possible explanations for the results of our study. Note that these are hypotheses which our study, lacking qualitative analysis, cannot confirm nor distinguish between; we view further investigation of these hypotheses as an opportunity for future work.

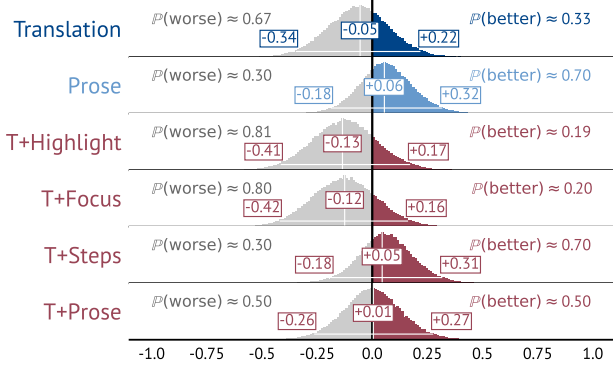
a) *Why was **CONTROL** better than the translation interfaces?*: The **CONTROL** interface offered a strict subset of information compared to all other interfaces. Yet, surprisingly, correctness often *dropped* when participants used the translation interfaces! Why might that be? One possibility is that, in



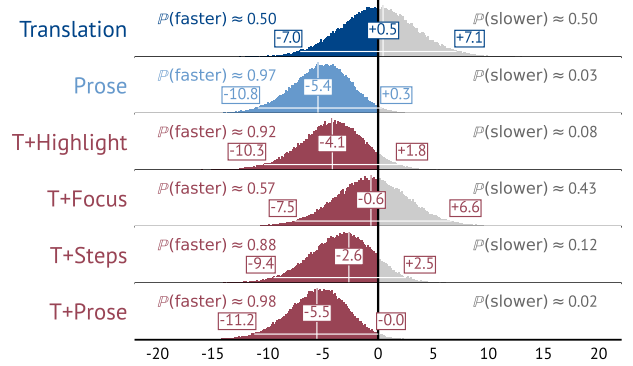
(a) Effect size for probability of success for Task 1.



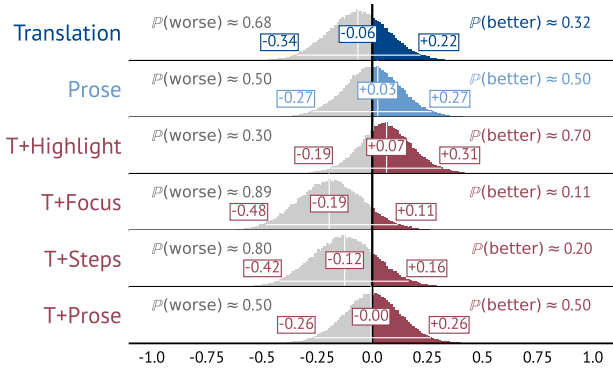
(a) Effect size for time taken among correct participants for Task 1.



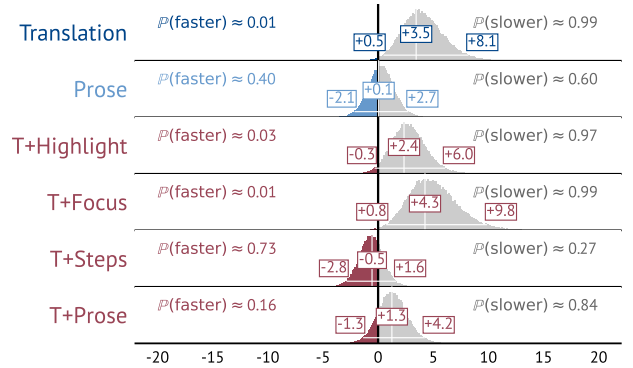
(b) Effect size for probability of success for Task 2.



(b) Effect size for time taken among correct participants for Task 2.



(c) Effect size for probability of success for Task 3.



(c) Effect size for time taken among correct participants for Task 3.

Fig. 7: **Posterior distribution of effect sizes (relative to CONTROL) for probability of success.** Effect size is calculated as difference of probabilities (ranging from -1 to 1). Better (higher probability of success) is to the *right*. The plotted interval is the 95% highest density interval (*i.e.*, the span of values that are most credible and cover 95% of the distribution). The measure of central tendency is the most credible value of the effect size (maximum a posteriori estimate).

Fig. 8: **Posterior distribution of effect sizes (relative to CONTROL) for time taken among correct participants.** Effect size is calculated as difference of median time taken in minutes. Better (less time taken) is to the *left*. The plotted interval is the 95% highest density interval (*i.e.*, the span of values that are most credible and cover 95% of the distribution). The measure of central tendency is the most credible value of the effect size (maximum a posteriori estimate).

the absence of a supportive interface, participants had to come face-to-face with their own lack of proficiency with the unfamiliar program and grapple with the complexities of the task head-on; in other words, it is possible that translation interfaces gave participants a false sense of security. Another possibility is that translation-based interfaces overloaded participants with too much information (a well-studied effect across various fields [77]), distracting them and lowering their performance. Cognitive Load Theory [78] suggests that an additional cost may lie in *split attention* between two representations of the code, requiring participants to reason simultaneously about both the original program and the translation and imposing extraneous cognitive load.

Another potential explanation is that translations did not provide the right kinds of information required to support our modification tasks. Prior work on information needs suggests that the value of an intervention depends on the fit between the questions a task generates and the information the intervention provides [54]. While translations may have answered questions about the behavior of the existing code, they might not have answered potentially more pragmatically important questions about how to use the unfamiliar library to create new behavior. A mismatch between the information provided by translations and the information needs of modification tasks may help explain our findings.

Yet another possibility is that the translation interfaces actively misled participants into believing falsehoods about their programs. From an analogical perspective, this may be consistent with Halasz and Moran’s [64] arguments about the drawbacks of analogy—that they may lead to false perceived correspondences between the base and target of the analogy. Participants given translations may have been misled into performing modifications that were valid in the familiar translation, but not in the unfamiliar starter code. Structure-mapping theory may explain the speedups observed for participants with augmented translation interfaces, but these participants, too, may have been subject to the same risks, leading to roughly comparable correctness between translation interfaces.

b) *Why was PROSE faster without altering correctness?:* We speculate that the PROSE interface recapitulated much of the information that participants using the CONTROL interface obtained via searching the web. In other words, participants may have received roughly the same information in the CONTROL and PROSE conditions, with PROSE eliminating the time spent searching for the information, leaving only the time required to understand it. This explanation would be consistent with the data we observed (faster times, roughly same rate of correct modifications), but does not account for the fact that the natural language explanations were tailored to the particular program at hand. In addition, Cognitive Load Theory might explain why the PROSE condition outperformed TRANSLATION, by integrating the original code with supplemental information instead of requiring participants to reason about separate code representations. Future work might investigate to what extent tailoring explanations to user programs helps them in working with unfamiliar code compared to documentation alone.

B. Implications for Design

More broadly, our results call into question what we should be designing *for* when we are designing an interface to support a technical task like program modification. Going into this study, we had the naïve perspective that such an interface should simply “teach” the user what the unfamiliar program does. The results from our study nuance and complicate this idea. Instead of “teaching” the user—a tall order for an automated interface—it may be worth considering more specific goals. For example, consider a possible explanation of our findings, that translation interfaces produce a false sense of security. If true, a key design priority in building translation interfaces would be to occasion the user to confront their lack of proficiency. This might require making interface design decisions that would otherwise be counterintuitive: increasing friction and reducing available information.

C. What affects translation effectiveness?

Our augmented translation interfaces allowed us to explore one possible factor among the set of factors that may influence translations’ effectiveness for program modification tasks: salience of specific translation features (*e.g.*, mapping between components). Beyond the representational salience we explored, there are many other features that may influence effectiveness. In particular, we suggest that future work should explore:

- Whether characteristics of the translation itself influence effectiveness (our work held the particular translations constant across conditions)
- Whether translations are more or less useful for certain domains and languages
- The role of participants’ programming expertise
- The role of participants’ domain knowledge or familiarity with *other* abstractions
- The role of task complexity
- Whether various cognitive factors (*e.g.*, problem-solving strategies) affect outcomes
- Whether translations are useful for other kinds of tasks, beyond program modification

Now that we have reason to question the assumption that translations are good explanations, we face an exciting open research problem around identifying and evaluating factors that influence translations’ usefulness for a variety of tasks.

VIII. CONCLUSION

In our 98-participant study tasking programmers with code modifications using an unfamiliar library, we found that participants with access to a translation were not substantially faster or less error-prone than those without access to a translation. Our results challenge traditional intuitions about translations and call into question what our goals should be for interfaces aiming to assist with practical program modification tasks. We hope that this work will serve as a resource for designers and researchers so that, as a community, we can design better tools to aid programmers in the increasingly common situation of working with code they did not write themselves.

REFERENCES

- [1] M. Lee, S. So, and H. Oh, "Synthesizing regular expressions from examples for introductory automata assignments," *SIGPLAN Not.*, vol. 52, no. 3, p. 70–80, oct 2016. [Online]. Available: <https://doi.org/10.1145/3093335.2993244>
- [2] I. Drosos, T. Barik, P. J. Guo, R. DeLine, and S. Gulwani, "Wrex: A Unified Programming-by-Example Interaction for Synthesizing Readable Code for Data Scientists," in *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. Honolulu HI USA: Acm, Apr. 2020, pp. 1–12. [Online]. Available: <https://dl.acm.org/doi/10.1145/3313831.3376442>
- [3] Z. Zhou, M. T. Tang, Q. Pan, S. Tan, X. Wang, and T. Zhang, "INTENT: Interactive Tensor Transformation Synthesis," in *Proceedings of the 35th Annual ACM Symposium on User Interface Software and Technology*, ser. Uist '22. New York, NY, USA: Association for Computing Machinery, Oct. 2022, pp. 1–16.
- [4] J. Cambrono, S. Gulwani, V. Le, D. Perelman, A. Radhakrishna, C. Simon, and A. Tiwari, "FlashFill++: Scaling Programming by Example by Cutting to the Chase," *Proc. ACM Program. Lang.*, vol. 7, no. POPL, pp. 33:952–33:981, Jan. 2023.
- [5] N. Shrestha, T. Barik, and C. Parnin, "It's Like Python But: Towards Supporting Transfer of Programming Language Knowledge," in *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct. 2018, pp. 177–185.
- [6] W. Crichton, "Human-Centric Program Synthesis," Sep. 2019. [Online]. Available: <https://arxiv.org/abs/1909.12281v1>
- [7] G. R. Bai, J. Kayani, and K. T. Stolee, "How Graduate Computing Students Search When Using an Unfamiliar Programming Language," in *Proceedings of the 28th International Conference on Program Comprehension*, ser. Icpce '20. New York, NY, USA: Association for Computing Machinery, Sep. 2020, pp. 160–171.
- [8] J. Hu, P. Vaithilingam, S. Chong, M. Seltzer, and E. L. Glassman, "Assuage: Assembly Synthesis Using A Guided Exploration," in *The 34th Annual ACM Symposium on User Interface Software and Technology*, ser. Uist '21. New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 134–148.
- [9] E. S. Wiese, A. N. Rafferty, and A. Fox, "Linking Code Readability, Structure, and Comprehension Among Novices: It's Complicated," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, May 2019, pp. 84–94.
- [10] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: <https://www.tensorflow.org/>
- [11] J. Scholtz and S. Wiedenbeck, "Learning second and subsequent programming languages: A problem of transfer," *International Journal of Human-Computer Interaction*, vol. 2, no. 1, pp. 51–72, Jan. 1990.
- [12] —, "Learning new programming languages: An analysis of the process and problems encountered," *Behaviour & Information Technology*, vol. 11, no. 4, pp. 199–215, Jul. 1992.
- [13] K. P. Walker and S. R. Schach, "Obstacles to Learning a Second Programming Language: An Empirical Study," *Computer Science Education*, vol. 7, no. 1, pp. 1–20, Jan. 1996.
- [14] M. Bower and A. McIver, "Continual and explicit comparison to promote proactive facilitation during second computer language learning," in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '11. New York, NY, USA: Association for Computing Machinery, Jun. 2011, pp. 218–222.
- [15] C. O'Brien, M. Goldman, and R. C. Miller, "Java tutor: Bootstrapping with python to learn Java," in *Proceedings of the First ACM Conference on Learning @ Scale Conference*, ser. L@S '14. New York, NY, USA: Association for Computing Machinery, Mar. 2014, pp. 185–186.
- [16] E. Tshukudu and Q. Cutts, "Semantic Transfer in Programming Languages: Exploratory Study of Relative Novices," in *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE '20. New York, NY, USA: Association for Computing Machinery, Jun. 2020, pp. 307–313.
- [17] —, "Understanding Conceptual Transfer for Students Learning New Programming Languages," in *Proceedings of the 2020 ACM Conference on International Computing Education Research*, ser. Icer '20. New York, NY, USA: Association for Computing Machinery, Aug. 2020, pp. 227–237.
- [18] Y. Kao, B. Matlen, and D. Weintrop, "From One Language to the Next: Applications of Analogical Transfer for Programming Education," *ACM Trans. Comput. Educ.*, vol. 22, no. 4, pp. 42:1–42:21, Nov. 2022.
- [19] E. Tshukudu, "Understanding conceptual transfer in students learning a new programming language," Ph.D. dissertation, University of Glasgow, 2022.
- [20] K.-C. Lu, S. Krishnamurthi, K. Fisler, and E. Tshukudu, "What Happens When Students Switch (Functional) Languages (Experience Report)," *Proceedings of the ACM on Programming Languages*, vol. 7, no. Icfp, pp. 215:796–215:812, Aug. 2023.
- [21] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. Se-10, no. 5, pp. 595–609, Sep. 1984.
- [22] T. A. Corbi, "Program understanding: Challenge for the 1990s," *IBM Systems Journal*, vol. 28, no. 2, pp. 294–306, 1989, conference Name: IBM Systems Journal. [Online]. Available: <https://ieeexplore.ieee.org/document/5387570>
- [23] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 971–987, Dec. 2006.
- [24] M. Hansen, R. L. Goldstone, and A. Lumsdaine, "What Makes Code Hard to Understand?" Apr. 2013, arXiv:1304.5257 [cs]. [Online]. Available: <http://arxiv.org/abs/1304.5257>
- [25] R. Minelli, A. Mocci, and M. Lanza, "I Know What You Did Last Summer - An Investigation of How Developers Spend Their Time," in *2015 IEEE 23rd International Conference on Program Comprehension*, May 2015, pp. 25–35, issn: 1092-8138. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/7181430>
- [26] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring Program Comprehension: A Large-Scale Field Study with Professionals," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 951–976, Oct. 2018, conference Name: IEEE Transactions on Software Engineering. [Online]. Available: <https://ieeexplore.ieee.org/document/7997917>
- [27] C. Bird, D. Ford, T. Zimmermann, N. Forsgren, E. Kalliamvakou, T. Lowdermilk, and I. Gazit, "Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools," *Queue*, vol. 20, no. 6, p. 35–57, Jan. 2023. [Online]. Available: <https://doi.org/10.1145/3582083>
- [28] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. Icse '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3597503.3608128>
- [29] K. Moratis, T. Diamantopoulos, D.-N. Nastos, and A. Symeonidis, "Write me this code: An analysis of chatgpt quality for producing source code," in *Proceedings of the 21st International Conference on Mining Software Repositories*, ser. Msr '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 147–151. [Online]. Available: <https://doi.org/10.1145/3643991.3645070>
- [30] M. Q. Feldman and C. J. Anderson, "Non-expert programmers in the generative ai future," in *Proceedings of the 3rd Annual Meeting of the Symposium on Human-Computer Interaction for Work*, ser. Chiwork '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3663384.3663393>
- [31] S. Nguyen, H. M. Babe, Y. Zi, A. Guha, C. J. Anderson, and M. Q. Feldman, "How beginning programmers and code llms (mis)read each other," in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, ser. Chi '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: <https://doi.org/10.1145/3613904.3642706>
- [32] M. X. Liu, A. Sarkar, C. Negreanu, B. Zorn, J. Williams, N. Toronto, and A. D. Gordon, "“what it wants me to say”: Bridging the abstraction gap between end-user programmers and code-generating large language models," in *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, ser. Chi '23. New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: <https://doi.org/10.1145/3544548.3580817>
- [33] B. Shneiderman, "Exploratory experiments in programmer behavior,"

- International Journal of Computer & Information Sciences*, vol. 5, no. 2, pp. 123–143, Jun. 1976.
- [34] R. Brooks, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, no. 6, pp. 543–554, Jun. 1983.
 - [35] S. Wiedenbeck, "Beacons in computer program comprehension," *International Journal of Man-Machine Studies*, vol. 25, no. 6, pp. 697–709, Dec. 1986.
 - [36] S. Letovsky, "Cognitive processes in program comprehension," *Journal of Systems and Software*, vol. 7, no. 4, pp. 325–339, Dec. 1987.
 - [37] N. Pennington, "Stimulus structures and mental representations in expert comprehension of computer programs," *Cognitive Psychology*, vol. 19, no. 3, pp. 295–341, Jul. 1987.
 - [38] J. Koenemann and S. P. Robertson, "Expert problem solving strategies for program comprehension," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. Chi '91. New York, NY, USA: Association for Computing Machinery, Mar. 1991, pp. 125–130.
 - [39] V. Fix, S. Wiedenbeck, and J. Scholtz, "Mental representations of programs by novices and experts," in *Proceedings of the INTERACT '93 and CHI '93 Conference on Human Factors in Computing Systems*, ser. Chi '93. New York, NY, USA: Association for Computing Machinery, May 1993, pp. 74–79.
 - [40] A. von Mayrhauser and A. M. Vans, "Program Comprehension During Software Maintenance and Evolution," *Computer*, vol. 28, no. 8, pp. 44–55, Aug. 1995.
 - [41] J. Lawrance, R. Bellamy, M. Burnett, and K. Rector, "Using information scent to model the dynamic foraging behavior of programmers in maintenance tasks," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. Chi '08. New York, NY, USA: Association for Computing Machinery, Apr. 2008, pp. 1323–1332.
 - [42] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?" in *Proceedings of the 34th International Conference on Software Engineering*, ser. Icse '12. Zurich, Switzerland: IEEE Press, Jun. 2012, pp. 255–265.
 - [43] W. Maalej, R. Tiarks, T. Roehm, and R. Koschke, "On the Comprehension of Program Comprehension," *ACM Transactions on Software Engineering and Methodology*, vol. 23, no. 4, pp. 31:1–31:37, Sep. 2014.
 - [44] J. Siegmund, "Program Comprehension: Past, Present, and Future," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 5, Mar. 2016, pp. 13–20.
 - [45] N. Kulkarni and V. Varma, "Supporting comprehension of unfamiliar programs by modeling cues," *Software Quality Journal*, vol. 25, no. 1, pp. 307–340, Mar. 2017.
 - [46] A. Alaboudi and T. D. LaToza, "Using Hypotheses as a Debugging Aid," in *2020 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Aug. 2020, pp. 1–9.
 - [47] J. Scholtz and S. Wiedenbeck, "Using unfamiliar programming languages: The effects on expertise," *Interacting with Computers*, vol. 5, no. 1, pp. 13–30, Mar. 1993.
 - [48] A. J. Ko and B. Uttl, "Individual differences in program comprehension strategies in unfamiliar programming systems," in *11th IEEE International Workshop on Program Comprehension*, 2003., May 2003, pp. 175–184.
 - [49] T. M. Shaft and I. Vessey, "Research Report—The Relevance of Application Domain Knowledge: The Case of Computer Program Comprehension," *Information Systems Research*, Sep. 1995.
 - [50] P. Abtahi and G. Dietz, "Learning Rust: How Experienced Programmers Leverage Resources to Learn a New Programming Language," in *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, ser. Chi Ea '20. New York, NY, USA: Association for Computing Machinery, Apr. 2020, pp. 1–8.
 - [51] A. J. Ko, B. A. Myers, and H. H. Aung, "Six Learning Barriers in End-User Programming Systems," in *2004 IEEE Symposium on Visual Languages - Human Centric Computing*, Sep. 2004, pp. 199–206. [Online]. Available: <https://ieeexplore.ieee.org/document/1372321>
 - [52] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, no. 6, pp. 703–732, Dec. 2011. [Online]. Available: <https://doi.org/10.1007/s10664-010-9150-8>
 - [53] T. D. LaToza, D. Garland, J. D. Herbsleb, and B. A. Myers, "Program comprehension as fact finding," in *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ser. Esec-fse '07. New York, NY, USA: Association for Computing Machinery, Sep. 2007, pp. 361–370.
 - [54] A. J. Ko, R. DeLine, and G. Venolia, "Information Needs in Collocated Software Development Teams," in *29th International Conference on Software Engineering (ICSE'07)*, May 2007, pp. 344–353, iSSN: 1558-1225. [Online]. Available: <https://ieeexplore.ieee.org/abstract/document/4222596>
 - [55] D. Kirk, M. Roper, and M. Wood, "Identifying and addressing problems in object-oriented framework reuse," *Empirical Software Engineering*, vol. 12, no. 3, pp. 243–274, Jun. 2007. [Online]. Available: <https://doi.org/10.1007/s10664-006-9027-z>
 - [56] Y. Oda, H. Fudaba, G. Neubig, H. Hata, S. Sakti, T. Toda, and S. Nakamura, "Learning to Generate Pseudo-Code from Source Code Using Statistical Machine Translation," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 574–584. [Online]. Available: <https://ieeexplore.ieee.org/document/7372045/?arnumber=7372045>
 - [57] W. Gad, A. Alokla, W. Nazih, M. Aref, and A.-b. Salem, "DLBT: Deep Learning-Based Transformer to Generate Pseudo-Code from Source Code," *Computers, Materials & Continua*, vol. 70, no. 2, pp. 3117–3132, 2021, publisher: Tech Science Press. [Online]. Available: <https://www.techscience.com/cmc/v70n2/44664>
 - [58] H. Cho and S. Lee, "Java2Pseudo: Java to Pseudo Code Translator," in *CEUR Workshop Proceedings*, ser. Iseapsec '22, 2022. [Online]. Available: <https://www.semanticscholar.org/paper/Java2Pseudo%3A-Java-to-Pseudo-Code-Translator-Cho-Lee/ce1774ad621bc5127e7a9e026761b86fec0a08e6>
 - [59] S. Kandel, A. Paepcke, J. Hellerstein, and J. Heer, "Wrangler: interactive visual specification of data transformation scripts," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Vancouver BC Canada: Acn, May 2011, pp. 3363–3372. [Online]. Available: <https://dl.acm.org/doi/10.1145/1978942.1979444>
 - [60] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann, "Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, Oct. 2015, pp. 3–12.
 - [61] M. Mayer, G. Soares, M. Grechkin, V. Le, M. Marron, O. Polozov, R. Singh, B. Zorn, and S. Gulwani, "User Interaction Models for Disambiguation in Programming by Example," in *Proceedings of the 28th Annual ACM Symposium on User Interface Software & Technology*, ser. Uist '15. New York, NY, USA: Association for Computing Machinery, Nov. 2015, pp. 291–301.
 - [62] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers, "Using an LLM to Help With Code Understanding," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. Icse '24. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 1–13. [Online]. Available: <https://dl.acm.org/doi/10.1145/3597503.3639187>
 - [63] M. L. Gick and K. J. Holyoak, "Analogical problem solving," *Cognitive Psychology*, vol. 12, no. 3, pp. 306–355, 1980, place: Netherlands Publisher: Elsevier Science.
 - [64] F. Halasz and T. P. Moran, "Analogy considered harmful," in *Proceedings of the 1982 conference on Human factors in computing systems - CHI '82*. Gaithersburg, Maryland, United States: ACM Press, 1982, pp. 383–386. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=800049.801816>
 - [65] D. Gentner, J. Loewenstein, and L. Thompson, "Learning and transfer: A general role for analogical encoding," *Journal of Educational Psychology*, vol. 95, no. 2, pp. 393–408, 2003, place: US Publisher: American Psychological Association.
 - [66] S. A. Douglas and T. P. Moran, "Learning text editor semantics by analogy," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems - CHI '83*. Boston, Massachusetts, United States: ACM Press, 1983, pp. 207–211. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=800045.801612>
 - [67] A. MacLean, V. Bellotti, R. Young, and T. Moran, "Reaching through analogy: a Design Rationale perspective on roles of analogy," in *Proceedings of the SIGCHI conference on Human factors in computing systems Reaching through technology - CHI '91*. New Orleans, Louisiana, United States: ACM Press, 1991, pp. 167–172. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=108844.108869>
 - [68] D. Gentner, "The Structure of Analogical Models in Science," *Tech. Rep.*, Jul. 1980, eRIC Number: ED193011.
 - [69] —, "Structure-mapping: A theoretical framework for analogy," *Cognitive Science*, vol. 7, no. 2, pp. 155–170, Apr. 1983. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/S0364021383800093>

- [70] J. Lubin, J. Ferguson, K. Ye, J. Yim, and S. E. Chasins, "Equivalence by Canonicalization for Synthesis-Backed Refactoring," *Proceedings of the ACM on Programming Languages*, vol. 8, no. PLDI, pp. 223:1879–223:1904, Jun. 2024.
- [71] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F. L. Aleman, D. Almeida, J. Altenschmidt, S. Altman *et al.*, "GPT-4 Technical Report," Mar. 2024, arXiv:2303.08774 [cs]. [Online]. Available: <http://arxiv.org/abs/2303.08774>
- [72] A. Blackwell and T. Green, "Notational Systems—The Cognitive Dimensions of Notations Framework," in *HCI Models, Theories, and Frameworks*. Elsevier, 2003, pp. 103–133. [Online]. Available: <https://linkinghub.elsevier.com/retrieve/pii/B9781558608085500058>
- [73] J. Feigenspan, C. Kastner, J. Liebig, S. Apel, and S. Hanenberg, "Measuring programming experience," in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. Passau, Germany: Ieee, Jun. 2012, pp. 73–82. [Online]. Available: <http://ieeexplore.ieee.org/document/6240511/>
- [74] P. Dragicevic, "Fair Statistical Communication in HCI," in *Modern Statistical Methods for HCI*, J. Robertson and M. Kaptein, Eds. Cham: Springer International Publishing, 2016, pp. 291–330.
- [75] M. Kay, G. L. Nelson, and E. B. Hekler, "Researcher-Centered Design of Statistics: Why Bayesian Statistics Better Fit the Culture and Incentives of HCI," in *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, ser. CHI '16. New York, NY, USA: Association for Computing Machinery, May 2016, pp. 4521–4532.
- [76] Stan Development Team, "Stan reference manual, v2.36.0," 2024. [Online]. Available: <https://mc-stan.org>
- [77] M. J. Eppler and J. Mengis, "The Concept of Information Overload: A Review of Literature from Organization Science, Accounting, Marketing, MIS, and Related Disciplines," *The Information Society*, vol. 20, no. 5, pp. 325–344, Nov. 2004. [Online]. Available: <http://www.tandfonline.com/doi/abs/10.1080/01972240490507974>
- [78] J. Sweller, "Cognitive load theory, learning difficulty, and instructional design," *Learning and Instruction*, vol. 4, no. 4, pp. 295–312, Jan. 1994. [Online]. Available: <https://www.sciencedirect.com/science/article/pii/0959475294900035>