

hacking your CAREER

Nadia Polikarpova
(with material from Loris D'Antoni)

Nadia Polikarpova



assistant professor since 2017

department of
Computer Science and Engineering
@ UC San Diego

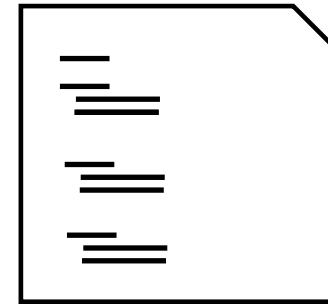
UC San Diego
COMPUTER SCIENCE & ENGINEERING

my research: program synthesis

specification



program



+ proof

my experience with NSF

CAREER: “Type-Driven Program Synthesis”

- submitted in 2019 (after year 2)
- awarded on first try

other grants: 2 small + 2 medium (+1 in linguistics)

reviewing: one panel (year 1) + ad-hoc

outline

1. before you begin
2. my process
3. what's inside

outline

1. before you begin

2. my process

3. what's inside

1. don't apply in your first year



2. volunteer to serve on panels



3. look at funded CAREER proposals



- proposals in your area are more relevant
- I asked 4 friends and based my structure on theirs
- since 2020, ~4 people asked to see mine
- Philip Guo (<https://pg.ucsd.edu/>) has annotated his proposal, email him to ask

4. choose topic you are known for



yes but not if you hate it

outline

1. before you begin
- 2.my process
3. what's inside

1. start early



may / june



Commits on Jun 22, 2019

Structure



nadia-polikarpova committed on Jun 22, 2019



Commits on May 30, 2019

Small changes



nadia-polikarpova committed on May 30, 2019

Initial commit (copy PGs template)



nadia-polikarpova committed on May 30, 2019



then I went to PLDI and
talked to some friends

july

Commits on Jul 14, 2019
Prelim results in 2.1 nadia-polikarpova committed on Jul 14, 2019
Commits on Jul 13, 2019
More refs nadia-polikarpova committed on Jul 13, 2019
Add a bunch of refs nadia-polikarpova committed on Jul 13, 2019
Some stuff in 2 nadia-polikarpova committed on Jul 13, 2019
Draft of 1.3 nadia-polikarpova committed on Jul 13, 2019
Draft of 1.2 nadia-polikarpova committed on Jul 13, 2019
Draft of 1.1 nadia-polikarpova committed on Jul 13, 2019
Commits on Jul 12, 2019
More in task 1 nadia-polikarpova committed on Jul 12, 2019
Start Task 1 nadia-polikarpova committed on Jul 12, 2019
Commits on Jul 11, 2019
Structure for background nadia-polikarpova committed on Jul 11, 2019
Commits on Jul 3, 2019
Integrate Ranjit's changes nadia-polikarpova committed on Jul 3, 2019
Commits on Jul 2, 2019
Minor nadia-polikarpova committed on Jul 2, 2019
Add pdf to git to help with feedback nadia-polikarpova committed on Jul 2, 2019
Polish intro a bit nadia-polikarpova committed on Jul 2, 2019
Add diagram, complete intro draft nadia-polikarpova committed on Jul 2, 2019
Commits on Jul 1, 2019
Most of intro draft nadia-polikarpova committed on Jul 1, 2019

Commits on Jul 15, 2019
Pass over 5.3 nadia-polikarpova committed on Jul 15, 2019
Draft of 8 nadia-polikarpova committed on Jul 15, 2019
Draft of 7 and summary nadia-polikarpova committed on Jul 15, 2019
Draft of education nadia-polikarpova committed on Jul 15, 2019
More in 5.3 nadia-polikarpova committed on Jul 15, 2019
More in 5.3 nadia-polikarpova committed on Jul 15, 2019
Commits on Jul 14, 2019
Some stuff in 5.3 nadia-polikarpova committed on Jul 14, 2019
Draft of 5.2 nadia-polikarpova committed on Jul 14, 2019
Draft of 5.1 nadia-polikarpova committed on Jul 14, 2019
Super rough draft of 4.1 nadia-polikarpova committed on Jul 14, 2019
Done with 2.1 nadia-polikarpova committed on Jul 14, 2019

Commits on Jul 17, 2019
Done nadia-polikarpova committed on Jul 17, 2019
Typos through sec 3 nadia-polikarpova committed on Jul 17, 2019
Done with intro nadia-polikarpova committed on Jul 17, 2019
Broader impacts nadia-polikarpova committed on Jul 17, 2019
Minor nadia-polikarpova committed on Jul 17, 2019
Minor nadia-polikarpova committed on Jul 17, 2019
Section 6 nadia-polikarpova committed on Jul 17, 2019
Done with 5.3 nadia-polikarpova committed on Jul 17, 2019
Done with 5.2 nadia-polikarpova committed on Jul 17, 2019
Add new plot nadia-polikarpova committed on Jul 17, 2019
Done with 5.1 nadia-polikarpova committed on Jul 17, 2019
Done with 4.2 and 7 nadia-polikarpova committed on Jul 17, 2019

2. get feedback

 Commits on Jul 3, 2019 

Integrate Ranjit's changes

 **nadia-polikarpova** committed on Jul 3, 2019

outline

1. before you begin
2. my process
3. what's inside

Context + gap + thesis
*What is the problem and
what is your core idea?*

Prior work / PI prep
*Why are you the right
person?*

Motivating examples /
applications
*Give reader something
concrete!*

Research Tasks
*What do you propose and
how are you going to
accomplish it?*

Evaluation plan
*What are the measures of
success?*

Education plan

Broader impact

Timeline

my structure

1. introduction
2. prior work
3. research tasks
4. education and outreach plan

my structure

1. introduction

- context + gap + thesis
- summary of research plan
- summary of education plan
- broader impacts

2. prior work

3. research tasks

4. education and outreach plan

1 Introduction

In recent years, *program synthesis* [32, 86, 89, 13] has emerged as a promising approach to automating low-level aspects of programming. A program synthesizer takes as input a high-level declarative *specification* of program behavior, and searches a large space of possible implementations to find one that satisfies the specification. Although there have been major advances in isolated domains, such as data wrangling and extraction [32, 95, 23, 41], optimizing low-level code [80, 68, 81], and feedback generation for programming assignments [84, 20, 33], making program synthesis part of mainstream software development process requires addressing two fundamental *challenges*:

1. *Specification*: How should the programmer communicate their intent to the synthesizer? Existing approaches use a variety of specification languages, from input-output examples to heavy-weight formal logics. A good specification language is *informative* (captures programmer intent well), *programmer-friendly* (easier to write than the program), and *synthesizer-friendly* (helps the search).
2. *Scale*: How do we scale program synthesis to real programming tasks? The biggest obstacle to practical program synthesis is the size of the search space a synthesizer has to explore, which grows extremely fast with the complexity of the solution and the number of available *components* (library functions the synthesizer is allowed to use). To tackle this challenge, we need better search algorithms as well as ways to seamlessly integrate synthesized code with hand-written code, in order to make synthesis more incremental.

Type-Driven Synthesis. Our approach to addressing both of these challenges is centered around *types*. The *thesis* of this proposal is that types can serve as a basis for specification languages and search strategies for program synthesis. Types are both high-level and precise, their programmer-friendliness is time tested, they can vary in expressiveness and capture both functional and non-functional properties, and finally, type checking is often fully automatic and compositional, which helps the search. Our thesis should not surprise anyone: in fact, we argue that static typing has been the single most successful programmer productivity tool in the history of software engineering. Not only do type checkers help prevent whole classes of programmer errors, types are also widely used for documentation, code completion (in tools like Microsoft IntelliSense and Eclipse Autocomplete), and API search (e.g. in Haskell’s API search engine HOOGL [60]). In this proposal, we simply push these ideas to their logical conclusion and explore different ways programmers can use types to communicate their intent.

context

gap

thesis

1 Introduction

In recent years, *program synthesis* [32, 86, 89, 13] has emerged as a promising approach to automating low-level aspects of programming. A program synthesizer takes as input a high-level declarative *specification* of program behavior, and searches a large space of possible implementations to find one that satisfies the specification. Although there have been major advances in isolated domains, such as data wrangling and extraction [32, 95, 23, 41], optimizing low-level code [80, 68, 81], and feedback generation for programming assignments [84, 20, 33], making program synthesis part of mainstream software development process requires addressing two fundamental **challenges**:

1. *Specification*: How should the programmer communicate their intent to the synthesizer? Existing approaches use a variety of specification languages, from input-output examples to heavy-weight formal logics. A good specification language is *informative* (captures programmer intent well), *programmer-friendly* (easier to write than the program), and *synthesizer-friendly* (helps the search).
2. *Scale*: How do we scale program synthesis to real programming tasks? The biggest obstacle to practical program synthesis is the size of the search space a synthesizer has to explore, which grows extremely fast with the complexity of the solution and the number of available *components* (library functions the synthesizer is allowed to use). To tackle this challenge, we need better search algorithms as well as ways to seamlessly integrate synthesized code with hand-written code, in order to make synthesis more incremental.

Type-Driven Synthesis. Our approach to addressing both of these challenges is centered around *types*. The *thesis* of this proposal is that types can serve as a basis for specification languages and search strategies for program synthesis. Types are both high-level and precise, their programmer-friendliness is time tested, they can vary in expressiveness and capture both functional and non-functional properties, and finally, type checking is often fully automatic and compositional, which helps the search. Our thesis should not surprise anyone: in fact, we argue that static typing has been the single most successful programmer productivity tool in the history of software engineering. Not only do type checkers help prevent whole classes of programmer errors, types are also widely used for documentation, code completion (in tools like Microsoft IntelliSense and Eclipse Autocomplete), and API search (e.g. in Haskell's API search engine HOOGL [60]). In this proposal, we simply push these ideas to their logical conclusion and explore different ways programmers can use types to communicate their intent.

goal: make your proposal easy to review

important stuff in bold

lists

named paragraphs

my structure

1. introduction

- context + gap + thesis
- summary of research plan (+ overview figure!)
- summary of education plan
- broader impacts

2. prior work

3. research tasks

4. education and outreach plan

if you remember one thing from this talk, it should be this:
make an overview figure

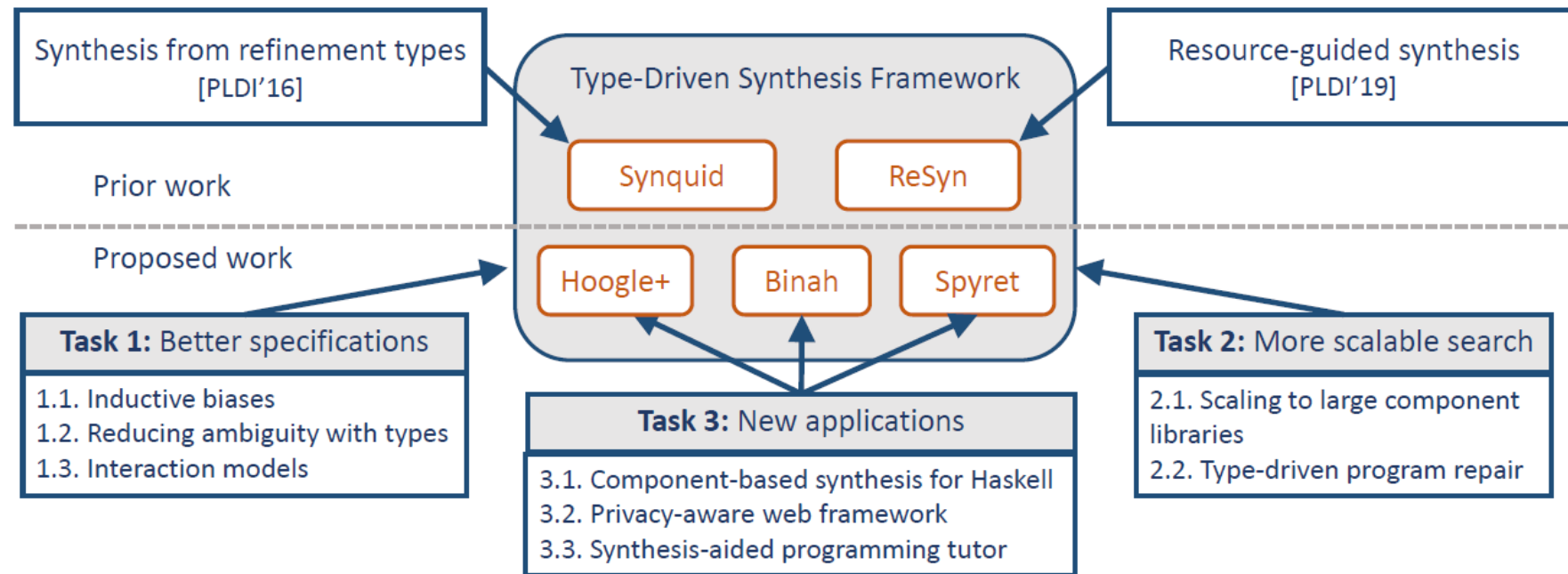


Figure 1: Overview of the research plan.

goal: make your proposal easy to review

1.1 Research and Education Plan

The overview of the research plan is shown in [Fig. 1](#). The research will be centered around the TYDRIS framework, and organized into three tasks outlined below.

Task 1: Better Specifications. Towards addressing the specification challenge, we will investigate novel uses of type systems as a means of communication between the programmer and the synthesizer. On the one hand, we will explore ways to make *existing* type annotations work better in the context of synthesis by equipping synthesizers with more natural inductive biases. On the other hand, we will develop new languages and user interaction models that enable the programmer to write concise yet informative specifications, and gradually refine them to obtain the desired result.

Task 2: More Scalable Search. Towards addressing the scalability challenge, we propose a new type-directed search algorithm we dub *type-guided abstraction refinement*. The main idea is to define a hierarchy of type systems with different levels of detail, and perform synthesis using *coarse-to-fine search*. This search strategy lets us quickly prune out completely irrelevant parts of the search space using coarse-grained type information, and then gradually hone in on the solution by refining the type information. As a complementary approach to scaling synthesis to practical programming tasks, we will develop a *type-driven program repair* technique, which uses type-driven synthesis to generate code patches.

Task 3: New Applications. Using the new techniques from the previous tasks, we will build three practical type-driven synthesis tools:

HOOGLE+ show-cases *synthesis in the small*. It is a component-based synthesis engine for Haskell, which extends HOOGLE [60] with the ability to synthesize code snippets (in addition to searching for individual API functions), and takes as input a Haskell type, optionally enhanced with refinements, examples, or keywords.

BINAH show-cases *synthesis in the large*. It is a privacy-aware web framework, where the programmer specifies each privacy policy once, as a type associated with a piece of sensitive data. BINAH uses type-driven program repair to automatically insert patches throughout the application to ensure that the policy is always satisfied.

SPYRET show-cases *synthesis for education*. We extend PYRET, a programming language designed for teaching programming, with synthesis capabilities to order to give students programming hints and help them write better tests and types.

hack 1: one of the tasks can be "applications":

- helps ground/motivate your ideas
- helps with evaluation plan

hack 2: one of the applications can be educational!



Context + gap + thesis

*What is the problem and
what is your core idea?*

Prior work / PI prep

*Why are you the right
person?*



Motivating examples /
applications

*Give reader something
concrete!*

Research Tasks

*What do you propose and
how are you going to
accomplish it?*

Evaluation plan

*What are the measures of
success?*

Education plan



Broader impact

Timeline

my structure

1. introduction
2. prior work
 - this proposal is about **you** and what **you** have done
 - don't be shy!
3. research tasks
4. education and outreach plan

Context + gap + thesis

*What is the problem and
what is your core idea?*



Prior work / PI prep

*Why are you the right
person?*



Motivating examples /

applications

*Give reader something
concrete!*



Research Tasks

*What do you propose and
how are you going to
accomplish it?*

Evaluation plan

*What are the measures of
success?*

Education plan

Broader impact



Timeline

my structure

1. introduction

2. prior work

3. research tasks

- tasks 1&2:
 - research question
 - related + preliminary work
 - proposed work
- task 3 (applications):
 - proposed work
 - evaluation plan

4. education and outreach plan

goal: make your proposal easy to review

3.1 Task 1.1: Inductive Biases

Research Question 1: *There might be many programs that satisfy an incomplete specification; how do we select or prioritize those that the programmer is most likely to be interested in?*

← make research questions stand out

Related work. One approach is to analyze large code bases and train statistical models that can be used to predict likely programs given a type or other syntactic information [77, 61, 52]. While statistical methods are very useful when searching for commonly occurring code snippets, we conjecture that they are less effective for uncommon programming tasks, general-purpose data structure libraries or rarely used libraries, especially in the context of functional languages, which have less redundancy in the code.

Preliminary work. The PI and her team implemented a prototype of HOOGLE+, and conducted preliminary experiments on 44 queries collected from HOOGLE [60], STACKOVERFLOW, and our own experience. For each query we generated five solutions, using a component library with 291 popular Haskell functions. For each solution, we manually evaluated whether it is *interesting* and whether it exactly matches our intent. We found that the sources of low-quality solutions could be classified into three categories: (1) Programs that discard their inputs, *e.g.* $\backslash x1\ x2 \rightarrow []$ and $\backslash x1\ x2 \rightarrow x2$. (2) Programs that always fail, *e.g.* containing head `[]` as a subterm. (3) Homogeneous result sets: sometimes each result individually is interesting, but they are too similar; the synthesizer fails to venture “further afield” to find the program we intended.

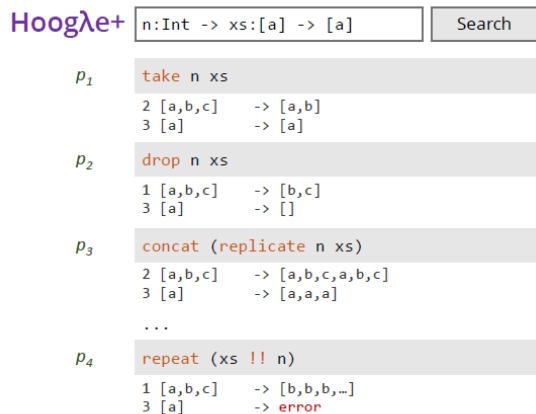


Figure 6: Mock-up of HOOGLE+.

```

1 -- | Conference.model:
2 Paper
3   title: String
4   conflicts: [User]
5   status: Status <!(viewer in elems (conflicts this))>
6
7 -- | App.hs
8 showAccepted = do
9   papers <- select (PaperStatus ==. Accept ? : Empty)
10  titles <- project papers PaperTitle
11  render ("Accepted papers:\n" ++ unlines titles)

```

Figure 7: Snippet from a conference management application in BINAH.

Intersection types. [27] explored a unifying type-theoretic framework, which casts examples as singleton types and combines them using *intersection types*; e.g. $([0] \rightarrow \text{False}) \cap ([-1] \rightarrow \text{True})$ is an intersection of two singletons. More generally, intersection types provide support for combining *partial specifications*, i.e. specifications of the function's behavior on a subset of its inputs. At the same time, SYNQUID-style refinement types can be used to express partial specifications concisely. For example, consider the function `allNeg` that takes a list of integers and tests whether all of its elements are negative. A complete specification for `allNeg` is hard to express in SYNQUID. On the other hand, a purely example-based approach would require many examples: inductive approaches are known to work poorly on Boolean functions, because their output contains very little information. However, a combination of refinement types and intersection types enables a concise yet highly informative specification:

$$\begin{aligned} \text{allNeg} :: (\text{Int} \rightarrow \text{Bool}) \cap & \left(\{ \text{Int} \mid \nu < 0 \} \rightarrow \{ \text{Bool} \mid \nu \} \right) \\ & \cap \left(\{ \{ \text{Int} \mid \nu \geq 0 \} \mid \text{len } \nu > 0 \} \rightarrow \{ \text{Bool} \mid !\nu \} \right) \end{aligned}$$

Here the first the first conjunct states that `allNeg` accepts any integer list, and the other two specify that `allNeg` returns true and false, respectively, if given a list of negatives and a non-empty list of non-negatives.

concrete examples
everywhere!

5.3 SPYRET: A Synthesis-Aided Programming Tutor

As our third application, we will integrate synthesis support into PYRET [74], a functional scripting language designed for programming education. PYRET is used to teach programming at several colleges and universities [4, 5, 6, 8] and in the Bootstrap [2] curricula, which integrate computer science into middle- and high-school education. The language is equipped with a web-based programming environment [9] and is part of a comprehensive teaching methodology [7] that promotes test-first, type-first programming.

Program synthesis and repair techniques have been used for education purposes before, most notably for grading and providing feedback on programming assignments [85, 20, 33, 12]. In this task, instead, we leverage PYRET’s web-based programming environment and the specification-first [2] programming style to give students real-time hints, help them improve tests, and to create multiple-choice exercises. We refer to the resulting, synthesis-aided version of the language and web environment as SPYRET.

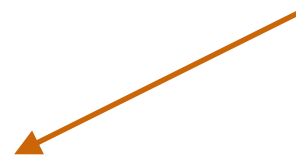
Programming hints. If a student is stuck while working on a programming assignment, they can ask SPYRET for help, and get feedback on whether they are on the right track and receive suggestions for what the next step might be. SPYRET will generate this feedback by attempting to synthesize completions for the student’s current solution, in such a way that they satisfy the teacher-provided specification. Similar functionality is offered by the on-line functional programming tutor ASKELLE [1], but they rely on higher-order unification instead of synthesis, and hence cannot provide any hints as soon as the student’s solution strays too far from the teacher’s solution. Using synthesis will enable SPYRET to give hints for a wide range of solutions, which is particularly important if students are allowed to define and use their own datatypes.

Improving specifications. Alternatively, we can use SPYRET to help students understand and improve their type annotations and tests. Assuming that the student starts solving a programming assignment by writing a (weak) specification, the idea is to repeatedly synthesize *incorrect* programs that satisfy the student’s specification, in order to expose its weaknesses. This is akin to using mutation testing [21] to assess the quality of a test suite, but does not require a complete program to mutate. The student will iteratively strengthen the specification until it is able to eliminate all the “mutants” (within a restricted search space). To develop such an *adversarial* synthesizer, we can invert the inductive biases we investigate in [Sec. 3], causing the synthesizer to prioritize programs that are likely incorrect.

Multiple-choice questions. This idea extends naturally to generating *multiple choice* drills. Given a specification and a correct solution provided by the teacher, SPYRET can generate *distractors*: subtly incorrect solutions to be used as alternative answers. An interesting research question is how to synthesize distractors that are incorrect in nontrivial and different ways. We can leverage programs previously generated by our adversarial synthesizer, since these programs violate properties that students did not think to test.

Evaluation plan. Joe Politz is one of the key members of the PYRET team and a colleague of the PI at UCSD. Together, Prof. Politz and the PI will integrate the synthesis features into the PYRET web environment and evaluate their benefits for college students. In particular, we will assess the effectiveness of the multiple-choice question generation technique for automated creation of *concept inventories* [76], a valuable yet traditionally time-consuming knowledge assessment tool.

evaluation plan clearly marked





Context + gap + thesis
*What is the problem and
what is your core idea?*



Prior work / PI prep
*Why are you the right
person?*



Motivating examples /
applications
*Give reader something
concrete!*



Research Tasks
*What do you propose and
how are you going to
accomplish it?*



Evaluation plan
*What are the measures of
success?*

Education plan



Broader impact

Timeline

my structure

1. introduction
2. prior work
3. research tasks
4. education and outreach plan

6 Education and Outreach Plan

The proposed research agenda is deeply integrated into the PI's education and outreach plan.

Curriculum Development. Since joining UCSD, the PI has been teaching the undergraduate *Programming Languages* class (CSE 130). This class introduces students to different programming paradigms, with a particular focus on functional programming and Haskell. We have observed that, coming from dynamic languages like Python and JavaScript, the students often struggle to understand, and more importantly *take advantage of* the Haskell type system. The PI has already introduced more in-depth material on type systems into this class. The availability of tools like HOOGLÉ+ will enable students to get a more interactive, hands-on experience with advanced types, demonstrating how types can work in their favor, not against them. The PI has also developed and taught a new graduate course on *Program Synthesis* (CSE 291). Several final project from the course have grown into conference submissions [15, 17]. All course materials are publicly available [70], and some have been reused by faculty at other universities (Prof. Shachar Itzhaky at Technion). CSE 291 already contains material on type-driven synthesis, which will be continuously updated and enriched based on the results of this project.

Education Tools. As detailed in [Sec. 5], this project will develop SPYRET, a synthesis-aided language and web-based environment primarily intended for teaching programming. We will work with the PYRET team and rely on their experience with CS education to integrate the tool into college courses that currently use PYRET [4, 5, 6, 8]. We expect SPYRET to benefit beginner students suffering from the “blank screen syndrome”, by giving programming hints, help students improving their type- and test-writing skills through adversarial synthesis, and make it easier for teachers to create multiple choice drills at scale.

Involving Undergraduates in Research. The UCSD Early Research Scholars Program (ERSP) was established to help broaden participation in research by targeting underrepresented groups earlier in their undergraduate career. ERSP provides team-based research apprentice experience for computer science and engineering majors in their second year of the program. Students work in teams of four, and each team is matched with an active research project. Students learn about research in computer science and then propose and carry out an independent research project over the course of an academic year. To date ERSP has provided early research opportunities for 138 students, including 96 female or non-binary students and 26 students from underrepresented racial and ethnic groups. Our analysis has shown that ERSP has many positive outcomes on participants, including overall higher grades, even than students with similar characteristics who applied to but did not participate in ERSP. The PI will mentor one ERSP team per year during all five years of this project, involving them into tool development and empirical evaluation. We have allocated explicit funding for a graduate student who will provide additional supervision to ERSP students, as well as conference travel funds for the program participants. Beyond ERSP, the PI routinely advises undergraduate research projects: during her two years at UCSD she has (co-)advised seven undergraduates.

Outreach. The PI has a track record of outreach. She co-organized the Programming Languages Mentoring Workshop (PLMW) at POPL'18 and personally raised money for travel grants for women and URM. She gave tutorials at both academic (PLDI'17) and developer (StrangeLoop'18) conferences, and has been a guest speaker or panelist at multiple events targeted at women or URM: a *Seminar on Women in Computer Science* at Ochanomizu University in Tokyo in 2015, MIT's *Women in Technology* program for high-school students in 2016, PLMW at ICFP'18, and the *HackXX* hackathon for women in 2019. She participated as a judge in two Student Research Competitions (at POPL'18 and PLDI'18). Finally, the PI is a member of her department's *Diversity, Equity and Inclusion* committee and a faculty co-advisor of the *Graduate Women in Computing* student group. The outreach efforts will continue with talks at the UCSD incoming freshmen seminar, Harvey Mudd college, and a tutorial at FMCAD in the Summer and Fall of 2019. Finally, the PI has always made an effort to have her research tools easily accessible through a web interface [3] to facilitate broad dissemination of research results, and will continue to do so with the tools developed in this project.

combine research & education

piggy-back on department & school programs
(don't forget to allocate funding!)

if you have track record in outreach,
use it!



Context + gap + thesis
*What is the problem and
what is your core idea?*



Prior work / PI prep
*Why are you the right
person?*



Motivating examples /
applications
*Give reader something
concrete!*



Research Tasks
*What do you propose and
how are you going to
accomplish it?*



Evaluation plan
*What are the measures of
success?*



Education plan



Broader impact

Timeline

lessons learned

1. (probably) wait until after second year
2. ask for help!
3. it's just another proposal
4. make your proposal easy to review