Meta: Enabling Programming Languages to Learn from the Crowd

Ethan Fast, Michael S. Bernstein
Stanford University
{ethan.fast, msb}@cs.stanford.edu

ABSTRACT

Collectively-authored programming resources such as Q&A sites and open-source libraries provide a limited window into how programs are constructed, debugged, and run. To address these limitations, we introduce *Meta*: a language extension for Python that allows programmers to share functions and track how they are used by a crowd of other programmers. Meta functions are shareable via URL and instrumented to record runtime data. Combining thousands of Meta functions with their collective runtime data, we demonstrate tools including an optimizer that replaces your function with a more efficient version written by someone else, an auto-patcher that saves your program from crashing by finding equivalent functions in the community, and a proactive linter that warns you when a function fails elsewhere in the community. We find that professional programmers are able to use Meta for complex tasks (creating new Meta functions that, for example, cross-validate a logistic regression), and that Meta is able to find 44 optimizations (for a 1.45 times average speedup) and 5 bug fixes across the crowd.

Author Keywords

programming tools; crowdsourcing; social computing

ACM Classification Keywords

H.5.3. Information Interfaces and Presentation: Group and Organization Interfaces

INTRODUCTION

Programs are strikingly redundant [6]. When creating new functionality, programmers often borrow code from community resources such as forum posts, Q&A sites, tutorials, and open source libraries [8, 34]. Despite their widespread influence on code, these resources are divorced from real programs. Programmers cannot look up how many people have used a code snippet from the web, examine how they used it, or track whether yet-to-be-written alternatives might become more appropriate for the task at hand [42].

Imagine you have answered a question on a community site such as StackOverflow [3]. We then copy the code snippet

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
UIST 2016, October 16-19, 2016, Tokyo, Japan
Copyright © 2016 ACM. ISBN 978-1-4503-4189-9/16/10 $15.00
DOI: http://dx.doi.org/10.1145/2984511.2984532

Count the vowels in a string **str** to **int** **60** users **2.162** ms

```
import re

@meta(parent="5700375c2f6a2f000330436a")
def count_vowels(s):
        return len(re.findall('[aeiou]', s, flags=re.I))
```

Warning: Meta has found a possible alternative that is ***1.3*** *times faster*

Example inputs:

```
count_vowels("UIST") #=> 2
```

```
count_vowels("CHI") #=> 1
```

Known errors:

```
count_vowels(['CHI', 'UIST']) #=> expected string
```

You can load this snippet with:

```
count_vowels = meta.load("http://www.meta-lang.org/5700a")
```

Figure 1. Meta is a domain specific language for Python that allows functions to track how they are used by programmers. Above, the *count_vowels* Meta page displays example inputs collected from real runtime traces, known errors the function has encountered, a link to an optimized version, and code to load and run the function in Python.

from your answer and modify it to make it faster. Today, even if we record your answer's URL in a comment in our code [7], you will not know that we are using your snippet, much less that we have made a faster version. Other people will not know either, unless we take the time to report it. And even if our program automatically reports our optimization [25], your code will not update itself with the improvement. This paper is about solving these distributed knowledge problems by integrating community resources directly into source code and broadening the data they capture.

We explore this idea through *Meta*: a language extension for Python that allows functions to track how they are used by a crowd of programmers. Each Meta function records runtime data like inputs and outputs and exceptions, how it is described, and who is using it. These functions are building blocks for new programming tools: an optimizer that replaces your function with a more efficient version written by someone else, an auto-patcher that saves your program from crashing by finding equivalent functions in the community, and a proactive linter that warns you when a function fails elsewhere in the community. Like other programming resources, Meta presents a website that helps you find solutions for common coding tasks (Figure 2). But Meta's website is filled with content and runtime information drawn from code tagged in other people's programs.

We have implemented Meta as a library that programmers can import into existing Python programs.[1] The *@meta* decorator takes an optional documentation argument to transform a function into a Meta function:

```
@meta("Current temperature in a city, in Celsius")
def get_temp_of_location(city, country, apikey : private):
  response = urlopen("api.openweathermap.org/...")
  return json.loads(response)["main"]["temp"]

get_temp_of_location("Tokyo", "JP", my_api_key) #=> 15.83
```

And *meta.load* loads an existing Meta function into a program through a link to its page on the community website:

```
normalize_to_one = meta.load("http://www.meta-lang.org/
     snippets/56fead85a8de27000309512b")

normalize_to_one([3.798, 3.448, 0.728])
  #=> [0.476224, 0.43238086, 0.09139514]
```

All functions annotated with *@meta* appear on the community website and can be loaded by any other programmer.

Once loaded, these functions expose a second set of APIs. For example, while Python is dynamically typed, a Meta function (such as *normalize*) can tell you its type signature by analyzing its inputs and outputs via *normalize.get_type()*, or show examples inputs and outputs via *normalize.examples()*. These low-level APIs enable more complex API methods, such as *find_duplicates*, which searches the community for other functions that behave similarly on its inputs. Together, these APIs enable new tools such as a *crowd optimizer*, which replaces a function with a faster equivalent written by someone else; or an *auto-patcher*, which keeps your program running by falling back to an equivalent function in the community when your version throws an exception.

Meta's community lives at **www.meta-lang.org**, where programmers can search for existing functions by dynamic attributes like the types of arguments they take, how fast they are, or how many people use them. The functions on Meta are authored and maintained by the crowd — anyone can use these functions or push new contributions. The community also supports other lightweight forms of curation, such as flagging function data as inaccurate (for example, a documentation string may be poorly worded, or an example input may be misleading). We bootstrapped an initial set of functions for Meta by seeding it with functions derived from the 4,000 most popular Python posts on StackOverflow, paid expert crowdworkers from Upwork translated code on these posts into Meta functions, then provided these functions with sample inputs to generate a starter set of run-time information that Meta will collect under real-world use.

In our evaluation, we first examine the challenges programmers encounter when using Meta. We hired seven programmers from Upwork to solve a machine learning task. All seven successfully completed the task, writing 17 new Meta functions and loading 6 unique others across 748 lines of code. In total, 26% of the task code was composed of Meta functions: for example, cross-validating a logistic regression or computing tf-idf vectors across a corpus. Next, we measured the run-time overhead of Meta. We found that Meta functions added little overhead to function calls (0.31ms on average; in comparison, sorting a list of 5000 words takes 27ms), with larger one-time costs for loading (47ms) and creating (133ms) Meta functions. To test community optimization, we ran the optimizer over all code in Meta's community. Meta identified 44 optimizations, where the average optimized function was 1.45 times faster than its alternative.

[1] Install in Python 3 via: *pip install metalang*

In this paper, we explore what is possible when curating programmer behavior is a design goal of the programming language itself. This curation empowers programming tools with a greater awareness of the communities that use them.

RELATED WORK

Meta inherits from a large ecosystem of programming tools, and draws insight from other work in program analysis, crowdsourcing, and data mining.

Programming tools. Meta draws on community resources to make programmers more effective. Programmers often copy-paste from web resources [8]. Drawing on this practice, tools like Blueprint embed code search directly into the IDE [7], linking the code to its web page of origin. This link can also be maintained by mapping lines of source code to web browser resources viewed while editing [20, 16, 22, 24, 14] or directly on to examples [36]. Taken together, this work seeks to embed relevant web resources in programmers' code. Meta shares this goal through code search and linked copies of code across its community. However, Meta also aims for the reciprocal goal of embedding programmers' code and runtime data back into the web resources: links are bidirectional and explicit (though *meta.load*), allowing the language to capture runtime behavior across a crowd of programmers.

Sharing programmer data with a central server allows Meta to aggregate user behavior. There are privacy issues associated with sharing such data [47, 24], but users will still choose to make the tradeoff when the benefit is sufficient [37]. HelpMeOut opened this opportunity by capturing novice programmers' code before and after the resolution of compile-time and runtime bugs [25]. Other tools have applied static and dynamic analysis to cluster student programming solutions [19], or provide feedback on variable names [18]. Higher-level information is also available on GitHub through metrics like forks and stars. Codex mined this community to help programmers identify idioms their IDE and catch possible bugs [13]. We take inspiration from this work's shared idea of emergent practice: in Meta, the functions available to programmers adapt to meet the needs of the crowd.

Meta draws on dynamic code instrumentation to generate the majority of its community data. Similar instrumentation has been used in programming tools such as the Whyline to help programmers resolve misconceptions about how a program works [27, 26, 23] or to generate documentation [31]. Meta operates on a similar insight: by collecting runtime data from a crowd of programmers, we can help the community better understand how to use a given snippet of code.

Meta also expands on existing ideas in commercial and open source tools. For example, Algorithmia allows users to publish and search for common functions across a community

[1], and package managers such as *pip* or *npm* allow developers to publish code that anyone can install and import [39, 2]. Unlike Meta, these resources do not attempt to learn from the behavior of the code they index.

Program analysis. Researchers have applied program analysis methods to generate test cases for programs [49], infer the types of variables and functions in dynamic languages [44], uncover bugs [9] and fix these bugs automatically [45]. These techniques inform Meta's design and implementation. For example, we use dynamic analysis to infer the types of Meta functions, and leverage the natural redundancy of a programming community to patch and optimize functions, a generalization of heuristics demonstrated in prior work [41].

Natural language programming. End-user programming tools have demonstrated that predefined script templates [17, 28] can enable users to directly describe their desired behaviors in the user interface. This technique can also map a small set of keywords into an API call [33], enabling sloppy programming via less structured forms of syntax [35], or mapping natural language onto system commands [4, 15]. Meta enables natural language search to help guide code search and optimization by drawing on the community's textual metadata authored for each function.

Crowdsourcing. Software development is increasingly interfacing with the crowd. Early on, researchers articulated how scripts and algorithms could guide crowds [32, 48]. Since then, the relationship has become more complementary, as crowds have begun writing programs. Crowds have engaged in program synthesis techniques [11], collaborated to write complex software [40], written and updated program functions via microtasks [29], annotated code with natural language descriptions [13], and generated tutorials from code examples [21]. In contrast to these crowds, which tend to be paid and on-demand, Meta presents a vision more in line with StackOverflow, where the work of a community is central to the programming language and feeds forward to enable programmer productivity.

SCENARIO: DEVELOPING WITH META

Meta draws from the work of a community of programmers to help programmers write new code. Here we describe Meta through a scenario in which you build a classifier that can predict the gender of a tweet's author.

Using existing code in the community

Your first goal in writing this program is to read some training data into your program. This data is in *tweets.tsv*, where the first column is the gender of an author and the second is the content of the tweet. You would like to convert these data into a list of lists, where each element of the outer list is another list that corresponds to the data split on columns.

You remember using code for this before, so you load the function based on a description of what it does:

```
from metalang import Meta
meta = Meta()

load_tsv = meta.search("read tsv into list of lists")
print(load_tsv("tweets.tsv"))
```

Some of your one-off scripts are almost entirely made up of calls to *meta.search*, but in this case you would prefer something permanent. When you run this code, Meta tells you:

```
Note: for "read tsv into list of lists" using load_tsv
http://www.meta-lang.org/snippets/56f84b97cd0a6300030392d7

[["getting brunch with @people", 0],
 ["why so #cloudy", 1], ...]
```

The print output confirms that this Meta function behaves as you thought. You double-check the function via the page at its URL (it is popular and has been called on data similar to your own) and so you lock the function in place through an explicit link. The code at this URL will never change.

```
tsv_to_dict = meta.load("http://www.meta-lang.org/snippets
     /56f84b97cd0a6300030392d7")
print(tsv_to_dict)
```

Patching your code with other community functions

Your tweet data is in unicode, and you know you'll need to convert it into ascii before you can process it further. You decide to write a helper function to make the conversion, and you add a *@meta* decorator to link the code back to Meta in case someone else later writes a faster or better version:

```
@meta("convert unicode to ascii")
def unicode_to_ascii(s):
  return s.encode("ascii")

unicode_docs = [unicode_to_ascii(row[0]) for row in data]
print(unicode_docs)
```

To your surprise, when you try to decode your tweet data Meta prints out the following warning:

```
Warning: Patched unicode_to_ascii with
http://www.meta-lang.org/snippets/56ee4504c0cb8f7470fbaf40
Saved from: UnicodeEncodeError: 'ascii' codec can't encode
      characters in position 1-5

["getting brunch with @people"
 "why so #cloudy",
 ...]
```

When your function failed, Meta caught the exception and patched it with a community equivalent (which behaves the same as your function on all successful inputs). You look at the web page of the replacement and realize you should have passed *encode* another argument, "ignore", that tells it to ignore unsupported character codes. You change your code to point to this new function instead. For its part, Meta notes your function's failed inputs on its community page.

Optimizing your code through the community

The next thing you need to do is vectorize your data to train a classifier. You have been working on some math-intensive programs lately, so you know how to write a few lines of code to convert your tweet data into a matrix that encodes your text features as bags of words.

```
@meta("encode list of documents as bag of words")
def encode_docs(docs):
  vocab = set(reduce(lambda x,y: x+y,
                    [d.split() for d in docs]))
  word_to_index = {w:i for i,w in enumerate(vocab)}
  matrix = numpy.zeros(len(docs),len(vocab))
  for i,d in enumerate(docs):
    for w in d.split():
```

```
        matrix[i,word_to_index[w]] = 1
    return matrix, vocab

encode_docs.optimize()
```

Has anyone written something faster? You make *encode_docs* a Meta function and tell it to optimize itself. This works by looking up functions with the same behavior on all their arguments but faster execution. And to your surprise, Meta reports that a faster version of your text encoding function is available. Reducing over a lambda function is an expensive way to flatten a list of lists, and this alternative version uses the *itertools.chain* function instead.

```
Note: Optimized encode_docs (line 3) with vectorize_text
See: http://meta-lang.org/snippet/9sd90fgd0ss223
Average time of 15ms vs 10ms
```

You decide to replace your code with the faster function.

Learning about downstream changes

You finish the task by looking up a function to train and cross-validate a logistic regression. A few months later you come back to the code, thinking you might port some of it to a new project analyzing the language of politicians. When you run your code again, Meta tells you that some of the functions you used have new and improved versions:

```
Warning: author of read_tsv has suggested a replacement
http://www.meta-lang.org/snippets/56f84b97cd0a6300030392d7
Description: "deal with header files"
```

You follow the link to the new snippet and discover the new version of *read_tsv* can identify and skip header lines in tsv files, which would otherwise break some modeling code. By chance, your new tsv file does have a header. So you change your *meta.load* statements to refer to the new link.

Throughout this project, Meta helped you find code, make that code faster, and fix bugs when they occurred.

LANGUAGE ARCHITECTURE

Meta is a language extension that makes Python aware of how people use it, drawing from programs written by its community to help individual programmers. Here we describe how we instantiated Meta as a domain specific language.

Functions in Meta

Functions are the basic unit of analysis in Meta. By default, every Meta function is publicly indexed and available to any programmer using the language. Meta connects these functions with natural language descriptions, tracks their inputs and outputs, and observes what libraries they depend on. The language can then use these details to enable new forms of code search, optimization, and testing.

Creating a Meta Function

Meta functions are defined via the *@meta* decorator:

```
@meta("count the vowels in a string")
def count_vowels(s):
    return len(re.findall('[aeiou]', s, flags=re.I))
```

In the code above, *count_vowels* is now a Meta function. The *@meta* decorator takes an optional string argument that represents a short snippet of documentation, which Meta uses to enable more advanced code optimization.

In Python, a decorator (instantiated using @) is a higher order function that takes the function declared below it and returns a new version of that function. Because decorators provide a clean way to add new properties and capabilities to functions, programmers (and researchers [46]) often adopt decorator syntax when building domain specific languages.

Loading an Existing Meta Function

Meta is designed for code sharing that connects programmers' code directly, as opposed to copying and pasting a function or importing a library. Loading another programmer's function is a primitive in the language, for example:

```
count_chars = meta.load("http://www.meta-lang.org/snippets
    /5700375c2f6a2f000330436a")

count_vowels("reviewers are fun") #=> 7
```

Above, the url passed to *meta.load* points to the public community webpage for *count_vowels*.[2] This page contains the function's source code and other documentation mined from its use throughout the community, like example inputs or bugs. Whenever a new Meta function is created, a page is created for it on the community website.

The Anatomy of a Meta Function

Meta functions are callable Python objects that enable a range of new interactions. The biggest difference between a Meta function and a Python function is that Meta functions are saved in a public database and will actively record their runtime behavior. To enable this form of distribution, Meta functions must use no global state besides library imports.

When a new function is created, Meta records: *(1)* the name of the function *(2)* an optional documentation string passed to the *@meta* decorator *(3)* an optional argument to turn off data recording *(4)* the source code of the function *(5)* an optional *parent* argument that signals the function is based on an existing Meta function *(6)* who created the function *(7)* an optional argument for importable libraries the function requires, otherwise Meta will attempt to infer these from the run-time behavior of code. Together, these properties allow any Meta function to be loaded dynamically from the database and used in another programmer's code.

Similarly, each time a Meta function is executed the language run-time will record: *(1)* the arguments passed to the function, unless the function has turned off data recording or the argument has a *private* annotation[3] *(2)* the return value of the function *(3)* the types associated with a function's arguments and return value *(4)* how long it takes the function to execute *(5)* the function's user.

Meta instruments functions to capture their arguments and return values when they are run. To minimize overhead, Meta samples this run-time information at a probability of $1/n$, where n is the number of times a function has been called over program execution. For example, once a program has called a function 100 times, Meta has a 1% chance of capturing that function's run-time information. Similar approaches

[2] A link to the *count_vowels* function: **http://www.meta-lang.org/snippets/5700375c2f6a2f000330436a**

[3] E.g., *def add_one(secret : private): return secret + 1*

have been shown to limit the overhead of other kinds of program instrumentation [30].

Function versioning

When you update the definition of a Meta function, Meta will index a new version of that function once it has successfully processed at least one input. Each version of a function has a separate URL, and programs can always rely on the function stored at a given URL to remain the same. To notify other programmers who are using your functions about a new function version, you can mark that version as an *improvement*, and specify a message that will be passed as a warning to programmers using older versions of the function. To link a new Meta function to an existing function (for example, as an improvement or modification), you can pass *@meta* a parent argument that references the existing function (Figure 1).

Limiting network overhead via a local cache

To reduce Meta's runtime overhead, we cache recently loaded functions on disk. When loading a function, Meta will first check for it in the cache, then try its remote server. This allows programmers to use Meta with no network access (assuming they've already used each function) and lowers the runtime cost of initializing functions by avoiding network latency. Similarly, instrumentation data and newly created Meta functions are also stored in this cache and batch uploaded on program exit to Meta's server.

An Overview of the Meta API

The Meta API exposes higher-level analyses on top of the run-time information contained in the community database. Here we explain these analyses, and show how they enable new interfaces for code search, optimization, and testing.

Run-time type inference

Knowing the type of a function can be helpful as a search constraint ("I want a function that returns an integer") and also as a form of documentation. The Meta instance method *get_type* infers a type for a function based on the run-time values that have passed through it from the community.

For example, if *split_string* is a Meta function:

```
split_string("I love reviewers")
    #=> ["I", "love", "reviewers"]
split_string.dymanic_type() #=> 'str -> List[str]'
```

To generate the dynamic type for a function, Meta takes the union of all types that have passed through it at run-time. For example, if Meta has seen a function take an integer and return a string, and also take a float and return a string, then the function's type would be: *Union[int, float]* → *str*. Meta also supports compound types , for example *List[int]*. To decrease the risk of garbage inputs, Meta uses input types provided by at least two unique users when computing a type signature.

We take a pragmatic stance on type analysis, where Meta supports just enough complexity to enable our envisioned interactions. Meta can reason about Python primitive types and recursive combinations of them (e.g., in list, tuples, or dictionaries) but does not support class based sub-typing or leverage other forms of static reasoning about code.

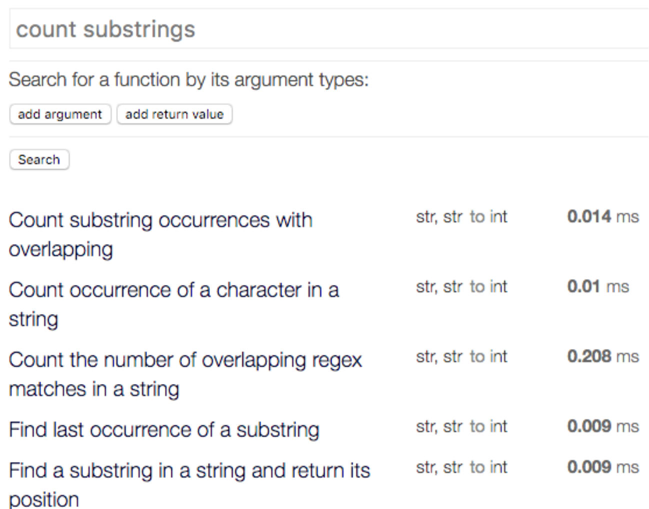

Figure 2. Meta's community website allows programmers for search for functions using dynamic attributes like the types of inputs it takes, example inputs, or how long a function takes to run.

Information about function types can enable many useful analyses [38]. Meta shows that dynamically typed languages can gain some of the benefits of function types by analyzing a function's execution across a crowd. These function types then enable many of Meta's more advanced APIs.

Inspecting example inputs and outputs

To understand what a function does and how it works, example inputs and outputs provide a useful form of documentation [7]. The *examples* API allows programmers to sample real arguments and return values for a function.

For example, if *extract_verbs* is a Meta function:

```
extract_verbs("go home") #=> ["go"]
extract_verbs.examples(3) #=>
  # [("He walked to the store", ["walked"]),
  #  ("I told him what I thought", ["told","thought"]),
  #  ("'run away', he said", ["run","said"])]
```

Above, the *examples* method returns argument and return value pairs for the *n* most common unique inputs to a Meta function across the crowd.

Existing programming resources sometimes provide examples of function behavior, but these examples must be created manually and may become out-of-date as code evolves. Meta's examples are data-driven (covering real inputs) and emerge as a positive externality from the community.

Searching for a Meta function

Many functions that programmers need write have already been written by someone else. Meta's *search* method helps you find these functions. For example, you can search on a function's documentation string:

```
search = meta.search("sort dictionary by values", n=2)

print(search) #=>
  [{"func": MetaFunction("sort dictionary by its values"),
    "popularity": 134, "execution_time": 0.0013
    "text_similarity":0.5 },
   {"func": MetaFunction("sort dictionary on keys"),
    "popularity": 150, "execution_time": 0.0011
```

```
    "text_similarity":0.3 }]

sort_dict = search[0]

sort_dict({"a":2,"b":1,"c":3"})
    #=> [('b',1),('a',2),('c',3)]
```

By default, results are ordered by string similarity, then popularity (the number of unique users who've used a function), and finally average run-time speed (in milliseconds). For convenience, if you don't pass *search* a number of results (e.g., $n = 2$) it will return the top function directly. You can also constrain a search by a function's inferred type signature:

```
rand_str = meta.search("generate random string by length",
                       type = "int -> str")
rand_str(3) #=> "g8v"
```

Or search on example inputs and outputs:

```
comma_split = meta.search(input="5,4,4,5,5", output=["5",
     "4", "4", "5", "5"])
comma_split("Reviewers, the best thing ever")
     #=> ["Reviewers", "the best thing ever"]
```

We implement *search* as a series of queries on top of Meta's database. For function descriptions, we add a text index to the database that generates a text similarity score based on the average number of words shared between a query and a function's documentation string after stemming and lemmatization. To match type constraints, we limit the results of a query to functions with types that meet the constraint. Finally, for example inputs and outputs, Meta first applies other constraints, then executes the remaining functions on the desired inputs, returning functions with matching outputs. We cache these execution queries in the database to prevent Meta from re-executing functions unnecessarily.

While *search* is directly accessible to programmers, its main point of access is Meta's community website (Figure 2). There, Meta provides a stable link to functions, via *meta.load*, that prevents searching for functions on every program execution. More broadly, Meta shows how code search can benefit from dynamic properties like example inputs and outputs or the speed or popularity of code.

Finding equivalent functions in the community

If Meta knows which functions do the same thing, it can optimize a programmer's code (for example, swapping out one function with another that computes the same value faster), or provide a fallback implementation for some functionality when the programmer's program throws an exception. The Meta instance method *find_equivalents* is a low level API that enables these analyses, returning other functions that provide equivalent behavior to the current function.

For example, if *cartesian_product* is a Meta function:

```
cartesian_product([1,2,3],[4,5,6])
    #=> [4,10,18]
cartesian_product.find_equivalents() #=>
  # [MetaFunction("take product of list"),
  #  MetaFunction("find vector product")]
cartesian_product.find_equivalents()[0]([1,2,3],[4,5,6])
  #=> [4,10,18]
```

To find these equivalent functions, the current function is compared to all existing functions that match its type signature and share at least one common word in their string descriptions (after lemmatization and removing stop-words). Take *cartesian_product* as an example. For each of its possible equivalents (let's call them *comp_func*), Meta compares the output of *comp_func* with that of *cartesian_product*, under all known unique inputs to *cartesian_product*. If all the outputs of both functions are the same, Meta labels *comp_func* as an equivalent of *cartesian_product*.

We have not solved the halting problem (or other problems that reduce to it). Functions that behave the same way on some inputs will not always behave the same way on future inputs. But if both functions have been tested on a large amount of data, they are likely to be the same [43]. Meta's idea of equivalency defines a probabilistic one-way "can replace with" relationship between functions. For example, if *list_product* is equivalent to *vector_product*, then *list_product* will return the same result as *vector_product* on all of the inputs to *vector_product*, but not the other way around.

All equivalents are listed on the function's web page on the Meta community web site, where other programmers can flag them in case of inaccuracy. While *find_equivalents* can be useful to programmers as they search for existing code, it's more critical as a component of Meta's advanced analyses.

Community-sourcing function optimization

There are always many ways to implement functions, and some ways will be faster or more efficient than others. The Meta function method *optimize* allows programmers to automatically replace a function with the fastest version available in the community. For example:

```
@meta.doc("sum the elements in a list of lists")
def sum_list(lst):
  total = 0
  for sublist in lst:
    for el in sublist: total += el
  return total

list_of_lists = [range(0,100000) for _ in range(0,10000)]

time(sum_list(list_of_lists)) #=> 'N seconds'

# Tell Meta to try to optimize this function
sum_list.optimize(True)

time(sum_list(list_of_lists)) #=> 'N-5 seconds'
```

When *optimize* is set to true, Meta will examine all equivalents of a function in the community, and redirect future calls to the fastest version available. This works by examining all equivalent functions under n repeated executions of a sample of m arguments ($n = m = 10$, by default) and determining whether one is significantly faster than the others (under a two tailed t-test). So in the second call to *sum_list*, Meta replaces it with a faster version that uses vector arithmetic instead of iterating over the lists. Meta also has an environment variable that, when set, tells the language to optimize all functions:

```
export META_OPT=True
python program.py
```

When optimizing functions, Meta produces logs that link to the new code:

```
Warning: Optimizing sum_list (line 3) with sum_list_numpy
See: http://meta-lang.org/snippet/41fdfsdj2333d33
Average time of 0.05ms vs. 0.02ms
```

Compilers often make code faster by translating low level operations into a more efficient form. Meta shows how similar transformations can apply to much higher level functions by drawing on a programming community. Enabling your code to speed itself up as other community members invent optimizations is one reason a programmer might add Meta annotations to their code.

Automatically patching run-time bugs

No matter how well-tested a program might be, code will occasionally encounter bugs when running in production. The Meta function method *auto_patch* enables a function to attempt to save itself after crashing by patching itself with an equivalent function from the community. For example:

```
@meta("compute logarithm of array elements")
def log_array(a):
  return [math.log(x) for x in a]

log_array([1+2j,2,0+1j,4]) #=> TypeError: can't convert
     complex to float

log_array.auto_patch(True)

log_array([1+2j,2,0+1j,4])
  #=> [(0.804+1.107j), (0.693+0j), 1.57j, (1.38+0j)]
```

When the programmer's function *log_array* fails on an array of complex numbers, Meta will recover from the error and try to use a community equivalent instead. In this case, another programmer has written a function that computes log with *cmath*, a function that supports complex numbers. Protecting a program from crashing is another reason why a programmer might annotate their code using Meta.

When *auto_patch* is set to true, Meta wraps the existing function in a try/catch handler that prevents the program from crashing on an exception. Instead, when a function raises an exception, this handler will look up the next-most popular equivalent function in the community (the function used by the most unique users) and try it on the original input. If no equivalent function exists or can successfully process the input, Meta will simply re-raise the trapped exception.

As with function optimization, Meta has a global flag that tells the language to *auto_patch* all functions:

```
export META_PATCH=True
python program.py
```

Similarly, Meta will log all successful patches:

```
Warning: Patched log_array with
http://www.meta-lang.org/snippets/56ee4504c0cb8f7470fbaf40
Saved from: TypeError: can't convert complex to float
```

Through Meta, the redundancy of the programming community provides a safety net for individual programs.

Community-sourcing bug reports

When you aggregate bug reports across the community, you make uncommon bugs more visible and increase the likelihood that bugs will be fixed. The Meta function method *bugs* returns all known bugs for that function.

For example, if *tfidf* (a common text processing function) is a Meta function:

```
tfidf("this is a doc",vocab)
    #=> {"this":0.001, "is":0.0002, "a":0.0001, "doc":0.5}

# What inputs has this function failed on?
tfidf.bugs() #=>
    # [{"input":"",
    #   "error":"IndexError: list index out of range"}]
```

Above, *tfidf* has failed when passed an empty string, something a programmer might want to fix (it should be possible, albeit bizarre, to compute term-frequency-inverse-document-frequency on an empty string). Further, as bugs become known to the community, Meta will let the programmer know through run-time warnings, for example:

```
Warning: New bug identified for "tfidf" (line 1) with
         http://meta-lang.org/snippet/b89fdII8d/bugs
```

To capture bugs, Meta intercepts any exceptions a function throws and records them in its database. Programmers can also mark other inputs as buggy on the community website. Filing bug reports is common practice on some public resources (a library on GitHub), but less common in others (an obscure StackOverflow post). Meta ensures that no obvious bug reports slip through the cracks.

Testing functions automatically

Testing functions automatically improves developer productivity and software reliability [49]. One useful approach is random testing: calling a function on random but correctly typed inputs and checking if it throws an exception [12]. However, random testing is a blind search over the space of possible inputs, and its bugs may or may not realistically occur in practice. The Meta static method *sample_type* extends this technique by generating test inputs from previously observed run-time values in the community. For example, we can use observed Response objects (from the *requests* library) to test a function that parses links from HTML:

```
import bs4
import requests

@meta("get links from a Requests html response")
def get_links(r):
  soup = bs4.BeautifulSoup(r.text, "html.parser")
  links = [a['href'] for a in soup.find_all("a")]
  return links

resp = meta.sample_type("Response")

for _ in range(0,2): get_links(resp())
  # => ["/", "/academic", ...]
  # => ["/index.html", "/about.html"]
```

Above, *sample_type* creates a function *resp* that will generate values of type *Response*. In the test loop, we use *resp* to sample two *Response* objects and pass them to our new function. By default, *sample_type* chooses values with a probability proportional to the number of times they've appeared in the community (a bias towards more likely values), but it can also invert this measure to sample uncommon values for more adversarial testing.

Haskell researchers have proposed a similar approach to automatic testing, randomly generating typed values under constraints [10]. In contrast, Meta benefits from an empirical understanding of common values, and uses this to guide its

selection of test inputs. Further, Meta can sample the community for complex types that are difficult to generate randomly (like borrowing a *LogisticRegression* object created by one programmer to test another programmer's code).

Dealing with Functions that use IO

Functions that use IO have side effects, like writing to the local filesystem or saving a record to a database, that are important to identify when doing code search and optimization. For example, if someone says "write a list to a file" they probably want code that uses IO, and if they say "return the list with the highest median element" they probably do not. To solve this problem, Meta gives functions that use IO a specific type:

```
@meta("write list to file, return elements written")
def write_list(fname,lst):
  with open(fname,'w') as f:
  for l in lst: f.write("{}\n".format(l))
  return len(lst)

write_list("test.txt",[1,2,3,4]) #=> 4
write_list.get_dynamic_type() #=>
     'str -> List[int] -> IO[int]'
```

The IO type prevents a function like *write_list* from coming up in a search result for a function that returns an integer (*int* doesn't match the type *IO[int]*) and allows programmers that want IO to specify that in their search. To prevent unwanted side effects on a programmer's system, Meta will not execute functions with an IO type under the *find_duplicates* API. To identify Meta functions with IO, we instrument a set of 52 core Python functions, including *open* and *write*. This instrumentation notifies Meta if any of these functions are called within a Meta function under dynamic type inference (for example, to read from a file). Meta adds an IO annotation to the types of these functions.

THE META COMMUNITY

Meta is a crowdsourced programming language, and so its community of online programmers is core to its success. Here we describe the community design space of Meta, and the decisions we have made for the current version of the platform.

Trusting programmer data

How should Meta index program run-time data, when not all of it can be trusted? On one end of the design space, Meta might index all data into its community, under the assumption that common use represents common practice, and every new datapoint is potentially valuable. On the other hand, Meta might instead build a more conservative kind of community, using only data from trusted programmers, because less experienced programmers may unintentionally misuse Meta functions and add noise to the dataset. Both designs have benefits and drawbacks. An inclusive community collects more data at the cost of introducing less trustworthy data into the system, and a conservative community collects higher quality data at the cost of potentially lower coverage.

Our current approach incorporates all programmer activity into its community. However, to decrease the likelihood of bad data (e.g., novice programmers misusing functions) biasing the system, Meta adds three constraints. First, all functions in the community index must have successfully processed at least one input in someone's program. Second, when inferring a function's type, Meta only considers types passed to a function by at least two unique users (this number can be raised as the community grows). Third, Meta provides a community flagging mechanism so that data relating to the function can be marked as inaccurate, like a misleading example, or an incorrectly labeled bug.

Function Versioning

How should Meta handle function versioning? A successful versioning system would allow functions to be updated (e.g., to fix a bug), stable over time (e.g., to avoid left-pad scenarios [5]), and usefully reasoned about. On one end of the design space, Meta might try to maintain one primary version per function. This would prevent redundant versions from being analyzed, and Meta could re-run old inputs on newer versions to avoid data sparsity. But what if a functions API changes? This would be problematic for analysis and dangerous for users of an older version.

On the other end of the design space, all versions of a function might be considered unique, and older versions would be available under search and optimization. Meta adopts this approach, giving each version a separate URL id. This means Meta must analyze more functions, so to constrain the analysis (and ignore any non-working versions), we only consider versions with at least two successfully recorded inputs. A user can mark a new version as an improvement to advertise it on the search pages of older versions, guiding the community towards newer code.

Ownership of Meta functions

Who owns a Meta function: the person who wrote it, or the language itself? Ideally, we don't want the author of a function to remove it from the community, as other programs might be relying on it to run. But it's also important to give authors credit for their functions and encourage the sharing of code. We give authors of Meta functions copyright under the MIT license. This assigns credit to contributing programmers but also grants Meta perpetual use of the code.

Crowdsourcing Meta's initial community

Bootstrapping a function set for Meta is critical to making it useful for its initial users. We used expert crowdsoucing [40] to load its database with functions from the 4,000 most highly viewed StackOverflow Python posts. Three expert crowdworkers from Upwork translated the best answer to each post to a Meta function and sample input. We asked that these translations be expressed as individual functions with no global state besides library imports. This translation process produced 2,500 Meta functions, which we use to inform the user study and analysis of overhead in Evaluation.

EVALUATION

What challenges do programmers face when using Meta? And how does Meta impact the performance of the programs they write? To find out, we studied how seven professional programmers used Meta in practice. We also ran two quantitative studies that measured, first, how much overhead Meta added to performance of the programs the programmers

wrote, and second, the extent to which Meta could apply optimizations to the functions across its dataset.

Studying how professional programmers use Meta

To better understand the issues programmers encounter when using Meta, we studied how professional programmers used the language to complete a machine learning task.

Method

We hired seven programmers from the Upwork platform to use Meta to complete a machine learning classification task. Specifically, we asked them to train a model that could use the docstring of a Python function to predict whether or not that function would return a list. For example, a function to "calculate variance of a list" doesn't return a list, but a function to "split a string on a delimiter" does return a list. We provided a ground truth dataset of 600 function descriptions, divided evenly between positive and negative examples, and also a link to a tutorial that explained how to use Meta.[4] After each programmer finished the task, we interviewed them to better understand the strengths and limitations of using Meta. We also examined their code to better understand how they applied Meta to the programming task.

Results

All seven programmers successfully completed the machine learning task with Meta. In total, these programmers wrote 748 lines of code, and 26% of all code was covered by Meta functions. Programmers created 17 Meta functions, loaded 6 unique Meta functions, and executed Meta functions 15,077 times. On average, they completed the task in 5 hours.

Almost all of the functions our study participants created would be useful to other people. For example, one programmer wrote a function to discover which features had maximum information, and another wrote code to encode word data as a vector. The Meta functions loaded by programmers tended to be utility functions like *bool_to_int* or *flatten_lists* or *iterate_file_lines*. In general programmers reacted positively to the language. "I'd like to use it in other projects that I do" (P3) and "I will most probably use it and also recommend to others" (P5). We had not asked whether they would use Meta in the future. P4 told us, "I usually find myself searching code repositories, such as github, and it would save me a lot of time browsing lines of code if they were annotated using Meta."

One challenge that programmers faced in using Meta was deciding at what level of abstraction they should use the language. Most programmers used Meta functions to wrap small bunches of functionality (like initializing a logistic regression model on certain parameters, then training it on some data). For instance, P3 told us Meta was best for "small functions, where there is little ambiguity as to what the function does." However, other programmers used Meta differently. P2 wrote his entire 54 line solution as one Meta function. "I could split the solution into multiple functions," he explained, "But I wrote it as only one function because all the code depends on the main pandas data frame." While such large, specialized functions may have value to the community as examples, they are less likely to be useful in other people's programs.

[4] `http://www.meta-lang.com/tutorial`

Another issue mentioned by programmers was security. "It's a hell of a security breach," P1 told us. "Certain companies wouldn't like it." P3 echoed that sentiment. "Really that url for the snippet might be switched out for malicious code." But programmers also had constructive suggestions. "I think Meta could ask for a quick code review and add 'sign' when the code is reviewed and secure," P2 said. "And if the code is not reviewed just add a small warning that it isn't reviewed yet." In this sense, Meta has a security profile similar to other platforms for remote code sharing like *pip* or *npm*. Programmers cannot change the source code at an existing Meta URL, so the only way to compromise a snippet is by exploiting the language infrastructure itself.

Similarly, most programmers had some worries about their privacy when using Meta. "What about privacy issues?" P1 asked. He told us the ability to mark some kinds of data as private was important. P2 agreed. "I think business owners won't like to copy their data to other servers. It would be better if there was an agreement to share the data only with developers and clients." P5 brought up a point of contrast, that "I feel privacy should be taken care of by the programmer, we should be careful when using Meta." When we mentioned Meta had a parameter to turn off data recording, all programmers agreed this was enough to address these concerns.

Programmers also wondered about the difficulties of building a community around Meta. "It needs to develop continuously so that community will grow. Meta should be popularized in certain circles, like scientific communities." P4 told us. Similarly, P3 raised questions about the difficulty of community curation. "I'm not sure how the function duplication problem will be dealt with. Inevitably there will be 50 split functions." We suggested Meta could identify these duplicates automatically, falling back on a "mark as duplicate" button like StackOverflow. Other programmers were more optimistic. "People like to be famous sometimes, that encourages participation" (P1) and "its simplicity will make Meta popular" (P5).

Measuring the overhead of Meta programs

Meta is built on instrumentation and network calls, so it is important to understand its run-time overhead. We measured the overhead of loading and calling Meta functions over the programs written by programmers in our first study.

Method

Using the seven programs submitted in our first study, we first measured the average time cost of loading a Meta function. For each Meta function, we then measured its average time to execute over its calls, and comparatively, how long its normal Python variant would take to execute over the same calls. We computed these measures with a full local cache (where Meta would not need to look up anything remotely), and also an empty cache, where every load would hit the remote server.

Results

We found the average time cost of loading a Meta function was 47 milliseconds (or 112 with a cache miss) and the cost of creating a function via *@meta* was 133 milliseconds (or 247 with a cache miss). These costs reflect the fact that Meta needs to evaluate a stored function's source code, or else save

it in the case of function creation. These were fixed costs that occurred only once per function per program execution. On average, actually calling a Meta function only added 0.31 milliseconds of overhead to a normal function call.

While Meta's function overhead was low, the majority of this overhead (64%) was caused by recording function arguments. Optionally, Meta can turn off this instrumentation, which would allow it to operate close to parity with non-Meta code. But enabling programmers to save time in writing and debugging bespoke solutions may outstrip any runtime wins or losses that the instrumentation produces.

Measuring the benefits of function optimization

To better understand the optimizations possible using Meta, we ran Meta's optimizer over all code in the community, then examined the distribution of performance improvements.

Method

We wrote a program that iterated over each Meta function in the community, first loading the function then calling Meta's *optimize* API, as described in our Implementation section. In summary, any optimized functions this API produces will 1) return the same values for all input arguments sampled from the original function 2) show statistically significantly improved run-time performance over the aggregate time cost of 10 calls to each sample argument. We then manually examined all suggested function optimizations to discover whether any were faulty (that is, the functions didn't actually do the same thing). Finally, we computed percentile statistics for the distribution of optimizations.

Results

In total, Meta found 44 correct optimizations. The median optimized function was 1.45 times faster than the non-optimized version. The interquartile range was 1.15-2.91 times faster.

Many of the most useful optimizations relied on an understanding of the performance characteristics of standard library functions. For example:

```
def merge_two_dicts(a, b):
  return dict(itertools.chain(a.items(), b.items()))

# this function is 2.4 times faster
def merge_dicts(d1, d2):
  d = d1.copy()
  d.update(d2)
  return d
```

We also found seven incorrect optimizations. Meta's mistakes were generally caused by data sparsity. For example, the function *is_capitalized* had only been observed on one run-time argument, "abc," which produced the result *False*. When "abc" was passed to a possible equivalent function, *str_is_number*, this second function also returned *False*, leading Meta to conclude that *str_is_number* could be used in place of *is_capitalized*. These mistakes disappeared when we added new run-time information to the system. For example, after we called *is_capitalized* on "Hello," Meta no longer suggested the faulty optimization.

Finally, when interpreting these results, we found five functions among Meta's suggested optimizations that could also serve as bug fixes. These functions tended to be more careful with their inputs, for example checking whether a number was greater than zero before computing a logarithm, or wrapping string conversion functions in a try/catch block.

LIMITATIONS AND FUTURE WORK

Meta presents a function-centric approach to building a crowdsourced programming language. Python programmers can accomplish many complex tasks using functions, but some tasks oriented around global state make less sense to encapsulate in this fashion (for example, a web application's routing function). In future work, we hope to expand Meta to better capture these stateful tasks.

To succeed, Meta needs to be able to scale. Publishing, loading, and searching for functions all scale horizontally in the current system. But function patching and optimizing rely on *find_equivalents*, which may become costly as the database expands. Meta automates its search for equivalent functions by running N candidates (e.g., functions with the same type signature and similar descriptions) on a sample of M inputs and checking if they return the same outputs. This is $O(N*M)$ and potentially expensive, even with a cache. But as the community grows, *find_equivalents* could instead become curated. A more human-centered strategy, for example, limiting a search for equivalents to functions with specific tags, or having the crowd mark functions as possible equivalents, would lower the computational burden.

Finally, we have already begun to build other capabilities into Meta. For example, could Meta's database of functions enable synthesis across its crowd? We might think of functions as typed puzzle pieces that Meta can fit together to meet new programmer needs. Our current prototype can synthesize new functions like "sort the words in a list" given a target type signature like *str* → *List[str]*. Applications like community synthesis might vastly increase the number of functions available to Meta's crowd of programmers.

CONCLUSION

In this paper, we show how connecting code across a community can enable programming resources that benefit from a much richer swath of runtime data and how tools can use these data to help programmers write better code.

Looking forward, Meta envisions a world in which an enormous amount of data about how people program is introspectable and available for analysis, making possible a new class of intelligent programming environments.

REFERENCES

1. Algorithmia. URL `https://algorithmia.com`.
2. npm. URL `https://www.npmjs.com/`.
3. Stackoverflow. URL `http://stackoverflow.com/`.
4. Adar, E., Dontcheva, M., and Laput, G. CommandSpace: modeling the relationships between tasks, descriptions and features. In *Proc. UIST*. 2014.
5. Ars Technica. Rage-quit: Coder unpublished 17 lines of javascript and "broke the internet". 2016.

6. Baker, B.S. On finding duplication and near-duplication in large software systems. In *Proc. Reverse Engineering 1995*. 1995.
7. Brandt, J., Dontcheva, M., Weskamp, M., and Klemmer, S.R. Example-centric programming: integrating web search into the development environment. In *Proc. CHI*. 2010.
8. Brandt, J., et al. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proc. CHI*. 2009.
9. Cadar, C., Dunbar, D., Engler, D.R., et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proc. OSDI*. 2008.
10. Claessen, K. and Hughes, J. Quickcheck: a lightweight tool for random testing of haskell programs. In *Acm sigplan notices*. 2011.
11. Cochran, R.A., et al. Program boosting: Program synthesis via crowd-sourcing. In *Proc. POPL*. 2015.
12. Duran, J.W. and Ntafos, S.C. An evaluation of random testing. In *Software Engineering*. 1984.
13. Fast, E., et al. Emergent, crowd-scale programming practice in the ide. In *Proc. CHI*. 2016.
14. Fourney, A., Lafreniere, B., Chilana, P., and Terry, M. InterTwine: creating interapplication information scent to support coordinated use of software. In *Proc. UIST*. 2014.
15. Fourney, A., Mann, R., and Terry, M. Query-feature graphs: bridging user vocabulary and system functionality. In *Proc. UIST*. 2011.
16. Fourney, A. and Morris, M.R. Enhancing Technical Q&A Forums with CiteHistory. In *Proc. ICWSM*. 2013.
17. Gao, T., et al. Datatone: Managing ambiguity in natural language interfaces for data visualization. In *Proc. UIST*. 2015.
18. Glassman, E.L., Fischer, L., Scott, J., and Miller, R.C. Foobaz: Variable name feedback for student code at scale. In *Proc. UIST*. 2015.
19. Glassman, E.L., et al. OverCode: Visualizing variation in student solutions to programming problems at scale. In *TOCHI*. 2015.
20. Goldman, M. and Miller, R.C. Codetrail: Connecting source code and web resources. In *Journal of Visual Languages & Computing*. 2009.
21. Gordon, M. and Guo, P.J. Codepourri: Creating visual coding tutorials using a volunteer crowd of learners. In *VL/HCC*. 2015.
22. Gordon, P.M. and Sensen, C.W. Seahawk: moving beyond html in web-based bioinformatics analysis. In *BMC bioinformatics*. 2007.
23. Guo, P.J., White, J., and Zanelatto, R. Codechella: Multi-user program visualizations for real-time tutoring and collaborative learning. In *Proc. VL/HCC*. 2015.
24. Hartmann, B., Dhillon, M., and Chan, M.K. HyperSource: bridging the gap between source and code-related web sites. In *Proc. CHI*. 2011.
25. Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S.R. What would other programmers do: suggesting solutions to error messages. In *Proc. CHI*. 2010.
26. Ko, A.J. and Myers, B.A. Designing the Whyline: a debugging interface for asking questions about program behavior. In *Proc. CHI*. 2004.
27. Kramer, J.P., Brandt, J., and Borchers, J. Using runtime traces to improve documentation and unit test authoring for dynamic languages. In *Proc. CHI*. 2016.
28. Laput, G.P., et al. PixelTone: a multimodal interface for image editing. In *Proc. CHI*. 2013.
29. LaToza, T.D., Towne, W.B., Adriano, C.M., and Van Der Hoek, A. Microtask programming: Building software with a crowd. In *Proc. UIST*. 2014.
30. Liblit, B., Aiken, A., Zheng, A.X., and Jordan, M.I. Bug isolation via remote program sampling. In *Proc. PLDI*. 2003.
31. Lieber, T., Brandt, J.R., and Miller, R.C. Addressing misconceptions about code with always-on programming visualizations. In *Proc. CHI*. 2014.
32. Little, G., Chilton, L.B., Goldman, M., and Miller, R.C. TurKit: Human Computation Algorithms on Mechanical Turk. In *Proc. UIST*. 2010.
33. Little, G. and Miller, R.C. Keyword programming in Java. In *Proc. ASE*. 2009.
34. Mamykina, L., et al. Design lessons from the fastest Q&A site in the west. In *Proc. CHI*. 2011.
35. Miller, R.C., et al. Inky: a sloppy command line for the web with rich visual feedback. In *Proc. UIST*. 2008.
36. Oney, S. and Brandt, J. Codelets: linking interactive documentation and example code in the editor. In *Proc CHI*. 2012.
37. Palen, L. and Dourish, P. Unpacking privacy for a networked world. In *Proc. CHI*. 2003.
38. Pierce, B.C. *Types and programming languages*. MIT press, 2002.
39. Python Software Foundation. pip. URL `https://pypi.python.org`.
40. Retelny, D., et al. Expert crowdsourcing with flash teams. In *Proc. UIST*. 2014.
41. Rinard, M.C., et al. Enhancing server availability and security through failure-oblivious computing. In *Proc. OSDI*. 2004.

42. Robillard, M.P. and Deline, R. A field study of API learning obstacles. In *Empirical Software Engineering*. 2011.

43. Sankaranarayanan, S., Chakarov, A., and Gulwani, S. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *Proc. PLDI*. 2013.

44. van Rossum, G. Mypy: Optional static typing for python. URL `http://mypy.readthedocs.org/`.

45. Weimer, W., Nguyen, T., Le Goues, C., and Forrest, S. Automatically finding patches using genetic programming. In *Proc. ICSE*. 2009.

46. Yang, J., Yessenov, K., and Solar-Lezama, A. A language for automatically enforcing privacy policies. In *Proc. POPL*. 2012.

47. Zhang, A.X., Blum, J., and Karger, D.R. Opportunities and challenges around a tool for social and public web activity tracking. In *Proc. CSCW*. 2016.

48. Zhang, H., Horvitz, E., Miller, R.C., and Parkes, D.C. Crowdsourcing general computation. In *Proc. CHI*. 2011.

49. Zhang, S., Saff, D., Bu, Y., and Ernst, M.D. Combined static and dynamic automated test generation. In *Proc. ISSTA*. 2011.