

Managing Ambiguity in Programming by Finding Unambiguous Examples

Kenneth Arnold Henry Lieberman

MIT Media Lab

{kcarnold, lieber}@media.mit.edu

Abstract

We propose a new way to raise the level of discourse in the programming process: permit ambiguity, but manage it by linking it to unambiguous examples. This allows programming environments to work with high-level descriptions that lack precise semantics, such as natural language descriptions or conceptual diagrams, without requiring programmers to formulate their ideas in a formal language first. As an example of this idea, we present Zones, a code search and reuse interface that connects code with ambiguous natural language annotations about its purpose. The backend, called ProcedureSpace, induces relationships between these purpose annotations, static code analysis features, and a variety of natural language background knowledge. ProcedureSpace can search for code given purpose descriptions or vice versa, and can even find code that was never annotated or commented. Since completed Zones searches become annotations, the system learns from user interaction. Users in a preliminary study found that reasoning jointly over natural language and programming language helped them reuse code.

Categories and Subject Descriptors H.5.2 [Information Interfaces and Presentation]: User Interfaces—Natural language; D.2.3 [Software Engineering]: Coding Tools and Techniques

General Terms assisted disambiguation, Blending

Keywords reuse, commonsense, SVD

1. Introduction

The process of authoring a program can be described as going from ambiguous representations about purpose and approach (mostly contained in the minds of the programmers) to unambiguous, highly structured representations of instructions

(mostly contained in computers). Development environments provide great assistance for the lowest level of this process: intelligent completion and refactoring greatly aid the input and verification of the resulting structured representation, and debuggers can help programmers determine why the structured representation behaves differently from their high-level purpose. Also, modern programming languages have gradually freed programmers from less relevant concerns and given them representational tools to guide their middle-level thinking. But most of the disambiguation process has remained the sole responsibility of the programmer.

Since the end products of programming should be as unambiguous as possible, many programming researchers have been understandably averse to allowing ambiguity even in high-level specifications. However, we suggest that this aversion may be unhelpful because it tends to force programmers to prematurely commit to some particular and precise way of thinking before they can dialogue with computers about their programs. Instead, we suggest that programming environments could permit programmers to describe the desired program in more natural terms, even if the terms themselves or the relations between them are ambiguous, then assist the programmer in the task of concretizing the program into an unambiguous executable form. We will call the interaction where a development environment assists a programmer in translating an ambiguous representation to a more concrete one *assisted disambiguation*.

1.1 Assisted Disambiguation Interactions

A development environment capable of assisted disambiguation could help programmers in several ways. First, it could help them find existing code that they could reuse in their programs. The programmer might specify the goal of the immediately desired code, or a subgoal may be inferred from a higher-level goal and the code written so far. Similar techniques could apply to other artifacts as well. For example, a goal could map to both to code and to behavioral tests that partially evaluate whether code accomplishes that purpose. Each represent part of the concrete specification of what it means for the code to satisfy the given goal. And inasmuch as failed test cases or other bugs indicate a failure of the con-

crete code to satisfy the ambiguous goal, we can also think of debugging this way. For instance, we could collect and analyze (situation, problem, solution) triples to learn what problems might come up in a certain situation and types of code changes that tend to fix them. Then we could make analogies to problem-solving strategies or concrete fixes that were helpful in similar situations. Finally, the programming environment could help the programmer document and distribute the result in a way that permits others to understand and utilize it.

In this paper, we focus on one initial example of assisted disambiguation: purpose-directed code reuse. In our interface, called Zones, programmers can associate a block of code (a Zone) with a natural language description of its purpose. By omitting either the purpose or the code, this annotation interface becomes a search interface, which becomes an annotation when the search is complete. The interaction is thus much like code search, but the search queries can include abstract characteristics that would not match a keyword in an identifier or type, and the retrieved code need not be commented. Also unlike a typical code search interaction, Zones learns how programmers describe purpose by capturing both successful and unsuccessful search interactions, and simultaneously learns what code characteristics are relevant to the goal and which are implementation details.

1.2 Backend Concerns

To assist in disambiguation, a programming environment must be able to relate ambiguous descriptions to unambiguous representations. One approach is to try various possible realizations of the ambiguous description; this approach has been successful where the ambiguous description straightforwardly defines a constrained space of possible unambiguous representations. Today’s large software libraries and open-source code repositories enable a different approach: learning relationships between ambiguous and unambiguous from examples. If a repository contains both code fragments and natural language descriptions of their purpose, the programming environment can learn how characteristics of the code relate to characteristics of their descriptions.

The backend of the Zones system, called ProcedureSpace, aims to relate two kinds of incomplete knowledge: purpose descriptions and code fragments. The knowledge about purpose descriptions is incomplete both because they are ambiguous to begin with and because the system has a very incomplete understanding of the natural language itself: the system may not have sufficient knowledge about a particular word used, or the words may be combined in an unfamiliar way. And knowledge about the code is incomplete in that it is generally not known what parts of the code are relevant to accomplishing the goal and what parts are merely implementation details. However, associations between code fragments and purpose descriptions disambiguate each other: ProcedureSpace, in effect, learns about purpose descriptions from the code they describe, and learns about code by studying how people de-

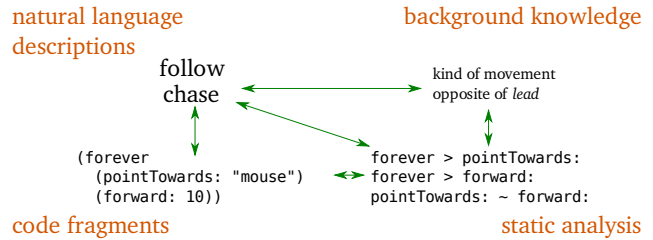


Figure 1. A representation of the words “follow” or “chase” that blends natural language and programming language

scribe its purpose. ProcedureSpace additionally uses semantic background knowledge about English words and phrases to help understand the natural language descriptions. The reasoning approach of ProcedureSpace is not formal logic but an application of a new technique called Blending [Havasi et al. 2009], enabling the easy use of a variety of different kinds of knowledge, so long as they can be made to overlap.

ProcedureSpace reasons jointly over, and induces relationships between, many different kinds of data: code purpose descriptions, static characteristics about the code itself, and natural language background knowledge, both general and domain-specific. As Figure 1 shows, ProcedureSpace understands words like “follow” and “chase” by relating them to commonsense background knowledge (such as “follow is a kind of movement”), examples of code that people have said causes something to chase something else, and common characteristics of the structure of that code. This paper does not seek to establish that these kinds of data are necessary and sufficient for a natural language code reuse system, but rather to show how these disparate kinds of data can be related by an automatic process.

1.3 Scratch

While assisted disambiguation is applicable to a broad variety of programming scenarios, we focus in this paper on novice programmers. Our environment for these experiments is Scratch, a graphical programming language designed for use primarily by children and teens, ages 8 to 16 [Resnick et al. 2003]. Scratch programming language components are represented by blocks that fit together like puzzle pieces to form expressions; stacking code blocks vertically causes them to execute in sequence. Scratch code is mostly comprised of concrete instructions that cause *sprites* to move around a *stage*, change their appearance, play sounds, or manipulate a pen, in response to various input including keyboard and mouse. The language includes control flow, Boolean, and mathematical statements, as well as sprite- or project-scoped variables. While the language lacks procedures or functions in the traditional sense, the event handler design makes concurrent modularity idiomatic and greatly simplifies task coordination. This makes a large amount of code reusable without significant modification. Though the specific techniques of this paper are somewhat tailored to the specific representa-

tions of Scratch, the general approach should be adaptable to other languages as well.

One main reason we chose to use Scratch for these initial experiments is the ready availability of a large quantity of potentially reusable code fragments. Scratch projects all have the same general structure, and project context has a limited effect on the behavior of individual lines of code, so Scratch code tends to be *a priori* more reusable, despite the general lack of software engineering discipline in the programmer community. The Scratch website [Monroy-Hernández and Resnick 2008] hosts over 300,000 projects, many already reusing code from other projects, all shared under a free software license. A quality-filtered sample of 6376 projects was used as the corpus for these experiments. Many of these projects are simple video games, so that domain will figure strongly in our examples.

2. Zones

Zones is our system for assisting disambiguation through purpose-oriented code reuse. A Zone is an active comment that associates a block of code with a brief natural language description of its purpose—“What’s this for?” Programmers can describe the purpose however they think about it; the description need not be precise, comprehensive, or even grammatical. There’s one recommendation: by only giving one line, the Zones UI encourages descriptions to be short. Figure 2a shows an example where a programmer has used a Zone to annotate a short section of Scratch code.

Omitting either the purpose or the code turns annotation into search. You can simply type an English statement of purpose, then the system searches for code that accomplishes that purpose (see Figure 2b). Alternatively, you can mark some code and then search, which means: find what purpose code like this generally serves (see Figure 2c).

Once an annotation or search is complete, it is added to the code fragment database, adding a new code fragment or a new way of explaining an existing one. The next programmer to look for related code fragments or purposes will benefit from the added code.

3. ProcedureSpace

How can we find code in a corpus given a natural language description of what purpose it should accomplish? The challenges facing this search task include differences in annotation word choice, variations in level of description detail, and sparsity of relevant annotations. ProcedureSpace, the backend for the Zones programming interface, takes a novel approach to this problem. It reasons jointly over code and words (or phrases) and incorporates additional information about how both the code and the words that apply to it may be related. The result is a representation that unifies syntactic knowledge about programs with semantic knowledge about goals. The ProcedureSpace technique not only improves answers to traditional code-search questions, but enables new types

of questions, such as: What annotations might apply to this code? What other code is similar in purpose to this?

ProcedureSpace works with six datasets that connect five different types of data: English concepts, relations between them, English purpose descriptions, code, and structural characteristics of that code. Table 3 summarizes these datasets.

3.1 Code Static Analysis

While many advanced techniques have been developed for static source code analysis, ProcedureSpace uses a deliberately simplified approach, focusing instead on combining static code analysis with natural language. The result will be akin to a quick glance at the code, rather than an in-depth study. The basic goal of the code analysis is similarity detection: two annotations might be similar if the code fragments they apply to are similar. For example, many different examples of code that handles gravity (or “falling”) all include a movement command conditioned on touching a color in the sky (or not touching a color on the ground). In other languages, such features could also include type constraints (e.g., “returns an integer”) or environmental constraints (e.g., “uses synchronization primitives”).

In extracting structural features, ProcedureSpace treats the code as a simplified syntax tree. For each code fragment, ProcedureSpace extracts various types of simple structural features about what kinds of code elements are present and how they are related:

Presence A particular code element is present somewhere in the fragment

Child A code element is the direct child of another code element

Containment A code element is contained within another code element, either as a parameter or as the body of a conditional or looping construct

Clump A clump of code elements occur in sequence

Sibling A particular code element is the sibling (ignoring order) of another code element

Within these types of features, it is not necessary to enumerate all possible features beforehand. Rather, for each feature type, an extraction routine generates all the features of its type that apply to a particular code fragment.

Consider the code fragment in Figure 3, which makes a sprite chase or follow another sprite. Table 3c shows examples of the code structural features extracted for the example code.

We then construct a matrix CS that relates code fragments to the structural features they contain. The rows of CS are the 14145 distinct code structure features that were extracted; the columns are the 127473 analyzed code fragments. (The order of the rows and columns does not matter for this kind of analysis.) An entry $CS(feature, fragment)$ is 1 if $fragment$ has $feature$, 0 otherwise. To keep long code fragments with large numbers of features from having a

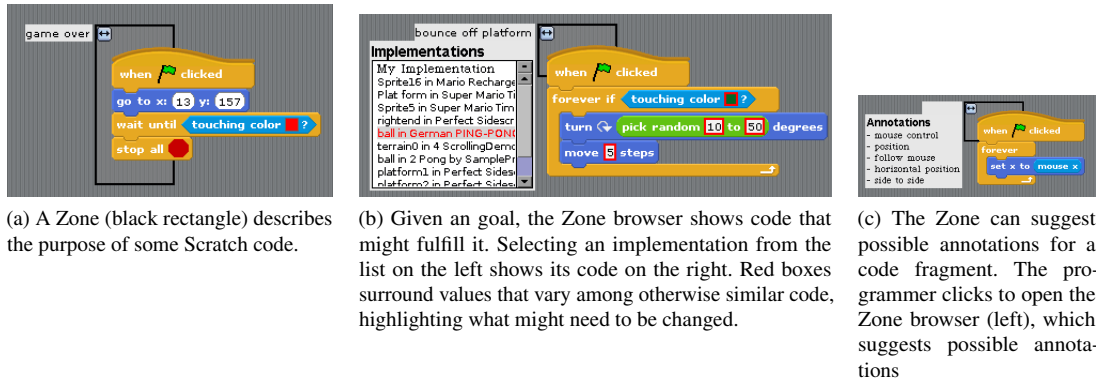


Figure 2. Types of interactions with Zones

Description	Rows		Columns		
	Kind	#	Kind	#	# Items
<i>CS</i> : code structure	structural features	14145	code fragments	127473	2721689
<i>AD</i> : annotations	purpose phrases	100	code fragments	126	174
<i>AW</i> : annotations as concepts	words	143	code fragments	126	429
<i>WC</i> : words in code	words	5639	code fragments	86519	208016
<i>DS</i> : domain-specific knowledge	words	19	English features	20	24
<i>CNet</i> : ConceptNet	words	12974	English features	87844	390816

Table 1. Descriptions and sizes of data going into ProcedureSpace in matrix form

(a) as presented in the Scratch UI

(FlagHat
(doForever
(pointTowards: "hero")
(forward: 1)))

(b) internal S-expression format

Child	doForever > pointTowards_
Containment	FlagHat doForever
Containment	FlagHat pointTowards_
Sibling	forward_ ~ pointTowards_
Clump	[forward_ pointTowards_]
Presence	pointTowards_

(c) Selected structural features

Figure 3. Example “chase” code fragment, taken from “Enemy AI tutorial: Chase.”

disproportionate effect on the analysis, we normalize all code fragments to have unit Euclidean norm. Here is the final matrix *CS*:

	when I receive Follow	when clicked	...
<i>CS</i> =			
Clump [forward_ pointTowards_]	0.27	0.27	...
Presence EventHatMorph	0.11	0	...
⋮	⋮	⋮	⋮

	when clicked	when clicked	when I receive Follow	...
<i>AD</i> =				
Purpose mouse control	1.00	0	0	...
Purpose chase	0	1.00	1.00	...
⋮	⋮	⋮	⋮	⋮

The purposes are actually stored as $(Purpose, purpose)$ tuples to distinguish the full strings from extracted words, which they will get mixed with later.

The initial annotations were entered by one of the authors. In various types of interactions with users, including the user study, other annotations were contributed.

3.2 Annotations


The Zones front-end provides annotations that link code fragments with a statement of purpose. A straightforward process encodes these annotations as a matrix: $AD(purpose, fragment)$ counts the number of times that *fragment* was annotated with *purpose*. The matrix was then tf-idf normalized, yielding a final matrix:

3.3 Annotation Words

Ideally, every code fragment that a programmer would ever want would be annotated exactly as he/she would describe it. But in practice, only a small fraction of code may ever be annotated, and the annotations rarely match exactly. This section, discusses two other ways to glean linguistic data for code fragments.

“Bounce off platform,” “BouncePlatform,” “platform bounce,” . . . are all different from the point of view of string equality, but you don’t need much background knowledge to know that they’re nonetheless highly semantically related. To help these line up, we extract *tokens* from each purpose description using standard natural language processing techniques. We then construct another matrix *AW* using the same process that constructed *AD*, i.e., $AW(token, fragment)$ counts the number of times that *fragment* was annotated with a purpose string that contained *token*. Since the tokens apply to the same code as the annotations that contain it, they will come out similar in the analysis unless they also apply to very different code. In either case, code that has related tokens will be pulled together.

The token extraction is done with standard natural language processing techniques: word splitting, case normalization, automatic spelling correction, lemmatization, and stopword removal. Each word of the result is treated as a token, but any two-word subphrase that also appears in the background knowledge sources (described below) is also included.



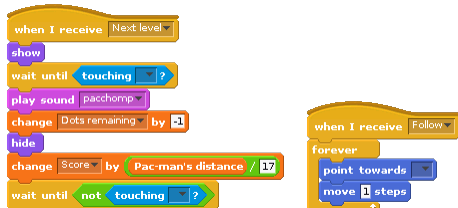
AW =

mouse	0.58	0.58	0.58	0	0	...
chase	0	0	0	0.71	0.71	...
⋮	⋮	⋮	⋮	⋮	⋮	⋮

3.4 Identifiers

Next, ProcedureSpace also relates words to the code fragments in which they occur, much like how a traditional search engine indexes a corpus of documents. Specifically, we extract natural language tokens from identifiers in the code fragments—names of variables and events that the programmer defined—to create a matrix *WC*. Tokens are extracted using the same procedure as in the previous section, including splitting underscore_joined and camelCased strings. An element $WC(token, fragment)$ counts the number of occurrences of *token* in *fragment*.

Here is a representative sample, again tf-idf normalized:



WC =

score	0.35	0	...
follow	0	7.93	...
⋮	⋮	⋮	⋮

3.5 Background Knowledge

When people choose entirely different words to describe their goals (and in user studies we found they often do), most search systems would be left with no relevant results. But ProcedureSpace incorporates background knowledge about how words relate.

One kind of knowledge is knowledge specific to the target domain. For Scratch, many projects are games, so helpful domain-specific knowledge includes facts such as “arrow keys are used for moving” and “moving changes position.” Such knowledge would enable us to relate an annotation about “arrow keys” with an annotation about “position,” for example. The matrix *DS* encodes a small, manually entered knowledgebase about simple games.

Another kind of knowledge is general world knowledge, such as “balls can bounce” and “stories have a beginning.” Without such knowledge, the system may be entirely unaware that an annotation of “bounce” may be relevant to find code for “moving the ball.” ConceptNet [Speer et al. 2008] provides a large database of broad intuitive world knowledge, expressed in a semantic network representation (e.g., *ball\CapableOf/bounce*). Rarely is a single ConceptNet relation a critical link in connecting two concepts; rather, the broad patterns in ConceptNet, such as which features typically apply to things that people desire or can do, help to structure the space of English concepts.

3.5.1 Matrix Encoding

Both ConceptNet and the domain-specific knowledge base are expressed as triples: *concept1\relation\concept2*. To form a matrix out of these triple representations, we use the approach of AnalogySpace [Speer et al. 2008]: for each triple, increment both (*concept1*, *relation\concept2*) and (*concept2*, *concept1\relation*). The columns of this matrix are called *features*. The double-encoding means that *arrow keys\UsedFor/moving*, for example, contributes knowledge about both “arrow keys” and “move.” For ConceptNet, connections that the community rated more highly are given greater weight; for the domain-specific knowledge, all entries are weighted equally as 1.0.

3.6 Blending

To reason jointly across these many different kinds of data, ProcedureSpace uses the Blending technique of [Havasi et al. 2009], which applies the dimensionality reduction methods of AnalogySpace to matrices of data that express relations of different kinds by taking advantage of the overlap between them. To apply the Blending technique to a set of matrices D_i , line up the labels (filling in zeros for missing entries) and add the matrices together:

$$A = \sum_i \alpha_i \text{MapLabels}(D_i)$$

where α_i is a weighting factor, set manually in these experiments. Then compute the Singular Value Decomposition (SVD), truncating to k singular values:

$$A \approx U_k \Sigma_k V_k^T$$

The matrices U_k and V_k represent each row and column of A as a vector giving its position along the k axes that represent

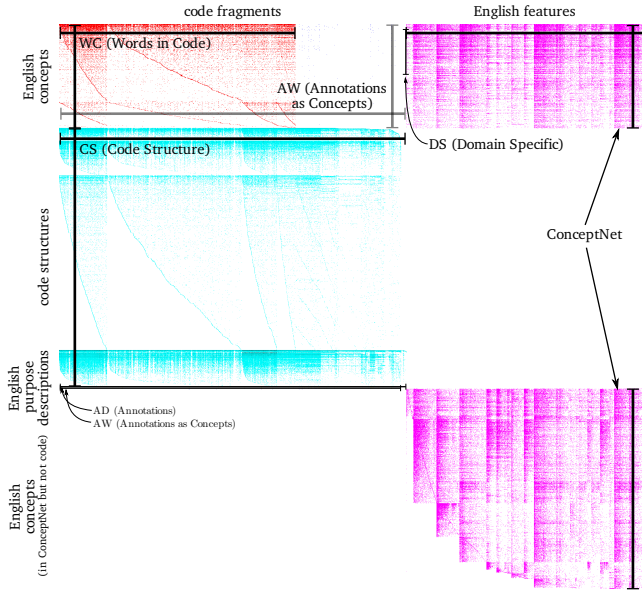


Figure 4. Coverage image for the ProcedureSpace matrix

the highest-variance dimensions of the data. The entries along the diagonal of Σ specify how important each axis is.

An important parameter for the Blending technique is the layout of the data matrices, since that determines which elements overlap and thus how they can be related. ProcedureSpace uses a novel layout, called the *transposed bridge blend* layout. *CS* relates code fragments to code structural features, while background knowledge relates English concepts to English features. The two domains would be entirely disconnected, except that the annotations link code fragments to English concepts. Figure 4 shows where each entry in the actual ProcedureSpace matrix comes from. Effectively, the purpose annotations bridge the structural features derived from static analysis with the natural language background knowledge in ConceptNet.

3.7 Goal-Oriented Search

Once we have used blending to construct ProcedureSpace, the search tasks required to power the Zones interface become straightforward vector operations. Each entity is a vector in the k -dimensional vector space: the U matrix gives the position of each English word, purpose phrase, code feature; the V matrix locates code fragments and English features. Since that all entities are in the same vector space, search operations can be expressed as finding vectors with high inner products. To find the vector \vec{p} of a query string composed of English words w_i , you simply sum the corresponding vectors:

$$\vec{p} = \sum_{i=0}^n U[w_i, :]$$

where the $:$ notation indicates a slice of an entire row. Then to find how well that description may apply to a particular code fragment, you take the dot product of \vec{p} and that code

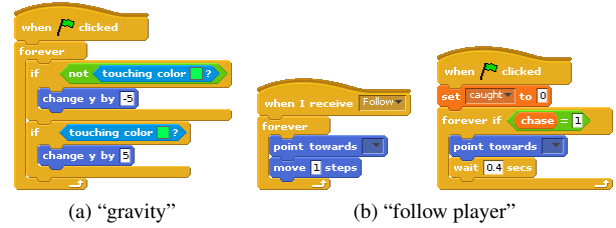


Figure 5. Sample search results

fragment’s vector (given by its row in V). In general, the weights for all code fragments are given by $V\vec{p}$, considering only rows of V that correspond to code fragments. The code fragments with the highest values are returned as the annotation search results, after filtering to remove code fragments that differ only in the values of constants.

Likewise, to find possible annotations for a code fragment, you extract its structural features f_i , form a vector $\vec{q} = \sum_{i=0}^n U[f_i, :]$, and find the words or annotations whose vectors have the highest dot product with \vec{q} . Or, to find which part of a given project performs a certain function, you compare the ProcedureSpace vector for the purpose description with the code fragments in that project.

3.8 Search Results

Users of our system searched for a variety of goals and expressed them in a variety of ways. Figure 5 shows selected results for some queries that users performed. The first search, “gravity,” returns a code fragment that was annotated “gravity,” illustrating that the indirect reasoning through code structure and natural language background knowledge rarely disturbs exact matches. For “follow player,” neither of the two results were exact annotation matches. The example code from Figure 3 was annotated “follow,” but the first result is one that matches both those code features and the word “follow.” The second match is very interesting because it inexactly matches at least two different kinds of data. The only common code structure is the presence of `pointTowards`, which evidently ProcedureSpace found to be associated with following. But many code fragments contain `pointTowards`; evidently this one was chosen because it also contained the word *chase*. Using a combination of common annotation data and background knowledge, ProcedureSpace related “follow” (the query word) and “chase,” and used this relationship to find a code fragment that is very different than what was annotated but nonetheless relevant.

4. User Experience

“Searching by goal is a really different way of programming,” said one participant in our preliminary user study. All participants in our two-task user study successfully used the Zones interface to find code that they could use in their project, and annotated both new and existing code in a variety of ways. We

were surprised by the number of different ways that people learned from their interactions with Zones.

The study was designed to address the question: *Does the Zones interface (both concept and implementation) help programmers make and use connections between natural language and programming language?* First, participants were familiarized with interacting with the Zones interface by annotating code and adding functionality to an example project. All participants were able to successfully use the Zones interface to annotate code.

The second part investigated how participants described behaviors that they observed by interacting, and evaluated whether they could (and would want to) use Zones searches to find code that performed those behaviors.

All participants were able to successfully imitate at least one behavior with the help of reused code from Zones. Finally, all participants left Zone searches as new annotations, validating our search-as-annotation paradigm.

Though participants reused some code exactly, much more frequently the code fragments would guide their thinking or point out Scratch functionality that they could use. One participant saw a *glide* (timed movement) command in a search result, and exclaimed: “Oh, it could be gliding... I forgot [about] the glide function.”

We were surprised by the number of different ways that people learned from their interactions with Zones. In one episode, a novice programmer corrected a flaw in his understanding of code while studying the search results for the annotation he was about to give it. A more experienced participant reported that the Zones interface forced her to think from a higher-level perspective. Frequently, participants appreciated learning something from seeing another person’s code, even if their goals were different or their understanding incomplete. And after the study, most participants took extra time to talk about how interesting the system and the underlying idea was to them.

5. Related Work

5.1 Code Search Systems

Code search systems can be distinguished by how programmers can query them. [Reiss 2009] includes a good survey of code search techniques, including formal specifications, type systems, design patterns, keywords, ontologies, tagging, and test cases. However, these code search systems have limited ability to reason about purposes that can be accomplished in a variety of ways, and their understanding of natural language is limited at best. ProcedureSpace uses annotations to reason about purposes and leverages both general and domain-specific natural language background knowledge.

A task switch away from development to even a good search engine can be distracting. [Fry 1997] and [Ye 2001] introduce the paradigm of reuse *within* development, linking code search into the IDE based on both comment keywords

and function signatures. Many systems are now integrated; a state-of-the-art example is Blueprint [Brandt et al. 2009].

Search-oriented systems like CodeBroker and Blueprint only directly benefit *consumers* of reusable software; users still have to publish their completed code manually. Zones makes it natural to share adapted or newly written code.

5.2 Programming in Natural Language

Natural language has often been seen as desirable as a high-level specification or programming language because it is a natural medium for communicating goals and ideas with a human collaborator. Various attempts have been made to interpret natural language as computer instructions directly, from COBOL to SQL to several modern attempts, including Pegasus [Knöll and Mezini 2006]. However, since programs must execute unambiguously, many previous attempts at natural language programming have required the use of unnaturally precise wording. Natural language representations of program present many challenges, but we think that managing ambiguity is a core challenge that has not yet received sufficient attention. Several projects have informed our thinking in this regard. Keyword Programming [Little and Miller 2007] matches keywords to commands and types in a function library. It is a useful tool for managing ambiguity on a low level: when a programmer know what keywords should appear in a line of code but not exactly how code is formed using these keywords, the Keyword Programming system can use search and type chaining techniques to disambiguate the keyword representation of that line of code. However, the programmer’s thinking must still be precise enough to use keywords. Metafor [Liu and Lieberman 2005] and its successor MOOIDE [Lieberman and Ahmad 2010] use sentence structure and mixed-initiative discourse to understand compound descriptions. MOOIDE further showed that general background world knowledge helps to understand natural language input. ProcedureSpace opens the possibility for these natural-language programming systems to *scale* by learning both statically from a corpus of code and dynamically through the Zones user interface. While Zones currently does not use natural language dialog, it could be very helpful, especially as we explore assisted disambiguation interactions that are longer than a single step.

5.3 Formal Specifications

There has been a lot of work in software engineering on the idea of formal specifications of code [Diller 1990; Hierons et al. 2009]. This body of work shares with us the idea of having some representation of the purpose of code at a higher, declarative level, that is independent from the code itself. It also has the ambition to provide algorithmic help to the programmer in assuring that the code meets the specification, or at least drawing the programmer’s attention to discrepancies between the two representations. But formal specifications are expressed in a mathematical language that most programmers find difficult to write. Such languages are

also entirely unsuitable for beginning programmers, which are our target user community here.

Our aim is not to assure the code does what the specifications say, but to give the programmer access to a body of alternative implementations of the specifications, and also the novel capability of reasoning backward from the code to specifications. By allowing programmers to express what may indeed be informal specifications in natural language, we hope to improve the accessibility of specifications as a programming methodology.

Logical reasoning systems used by formal specification tools have the advantage that they provide mathematical assurance of the validity of implementations. The downside is that their reasoning is not applicable to many practical programming problems. The Commonsense inference used by our system is good at making plausible inferences efficiently across a wide variety of programming situations, but it provides no guarantee of correctness.

6. Conclusion

Some think that programming is a constant war whose goal is to banish ambiguity from computer interaction. We think that ambiguity, especially of natural language, plays a vital role in the programming process, so programming environments should help programmers work with it. Our suggested interaction paradigm, *assisted disambiguation*, could help programmers to better understand their intent for a program and to manage the relationship between their intent and the more precise representations necessary for the computer to realize their intentions. Representing intent in natural language helps programmers avoid premature commitment to low-level decisions about procedures and representation. Our aim is to call a truce between natural language and programming languages.

We illustrate our suggestions with a prototype purpose-oriented code reuse system. Its frontend, called *Zones*, demonstrates an integrated interface for connecting natural language with Scratch code fragments to make comments that help programmers find and share code. The *ProcedureSpace* backend demonstrates important concepts in how to relate ambiguous and unambiguous representations as it reasons jointly over static code analysis, *Zones* annotations, and background knowledge to find relationships between code and the words people use to describe what it does.

Marvin Minsky said, “If you understand something in only one way, you don’t understand it at all”. We believe that understanding programming both by natural language descriptions and by programming language code, we can come closer to really understanding what we want computers to do.

Acknowledgments

We thank Andrés Monroy-Hernández and Rita Chen for providing the parsed Scratch projects, Rob Speer and Catherine

Havasi for helping with the Blending technique, and all our user study participants for their valuable input.

References

- J. Brandt, M. Dontcheva, M. Weskamp, and S. R. Klemmer. Example-centric programming: Integrating web search into the development environment. Technical report, CSTR-2009-01, 2009.
- A. Diller. *Z: An Introduction to Formal Methods*. John Wiley & Sons, Inc., New York, NY, USA, 1990. ISBN 047192489X.
- C. Fry. Programming on an already full brain. *Commun. ACM*, 40(4):55–64, 1997. ISSN 0001-0782. doi: <http://doi.acm.org/10.1145/248448.248459>.
- C. Havasi, R. Speer, J. Pustejovsky, and H. Lieberman. Digital Intuition: Applying common sense using dimensionality reduction. *IEEE Intelligent Systems*, July 2009.
- R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Comput. Surv.*, 41(2):1–76, 2009. ISSN 0360-0300. doi: <http://doi.acm.org/10.1145/1459352.1459354>.
- R. Knöll and M. Mezini. Pegasus: first steps toward a naturalistic programming language. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 542–559, New York, NY, USA, 2006. ACM. ISBN 1-59593-491-X. doi: <http://doi.acm.org/10.1145/1176617.1176628>.
- H. Lieberman and M. Ahmad. Knowing what you’re talking about: Natural language programming of a multi-player online game. In M. Dontcheva, T. Lau, A. Cypher, and J. Nichols, editors, *No Code Required: Giving Users Tools to Transform the Web*. Morgan Kaufmann, 2010.
- G. Little and R. C. Miller. Keyword programming in Java. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 84–93, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-882-4. doi: <http://doi.acm.org/10.1145/1321631.1321646>.
- H. Liu and H. Lieberman. Programmatic semantics for natural language interfaces. In *CHI '05: CHI '05 extended abstracts on Human factors in computing systems*, pages 1597–1600, New York, NY, USA, 2005. ACM. ISBN 1-59593-002-7. doi: <http://doi.acm.org/10.1145/1056808.1056975>.
- A. Monroy-Hernández and M. Resnick. Empowering kids to create and share programmable media. *interactions*, 15(2):50–53, 2008. ISSN 1072-5520. doi: <http://doi.acm.org/10.1145/1340961.1340974>.
- S. P. Reiss. Semantics-based code search. In *ICSE '09: Proceedings of the 2009 IEEE 31st International Conference on Software Engineering*, pages 243–253, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3453-4. doi: <http://dx.doi.org/10.1109/ICSE.2009.5070525>.
- M. Resnick, Y. Kafai, and J. Maeda. A networked, media-rich programming environment to enhance technological fluency at after-school centers in economically-disadvantaged communities. Proposal to National Science Foundation, 2003.

R. Speer, C. Havasi, and H. Lieberman. AnalogySpace: Reducing the dimensionality of common sense knowledge. *Proceedings of AAAI 2008*, October 2008.

Y. Ye. *Supporting Component-Based Software Development with Active Component Repository Systems*. PhD thesis, University of Colorado, 2001.