Managing Ambiguity in Programming by Finding Unambiguous Examples

Kenneth C. Arnold   Henry Lieberman
MIT Media Lab, MIT Mind Machine Project
20 Ames Street, Cambridge, MA 02139 USA
{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 informal 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 statements about its purpose. The backend, called ProcedureSpace, relates purpose statements, static code analysis features, and natural language background knowledge. ProcedureSpace can search for code given statements of purpose or vice versa, and can find code that was never annotated or commented. Since completed Zones searches become annotations, system coverage grows with 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 Design, Human Factors

Keywords ambiguity, Blending, common sense, informal representation, natural language, reuse

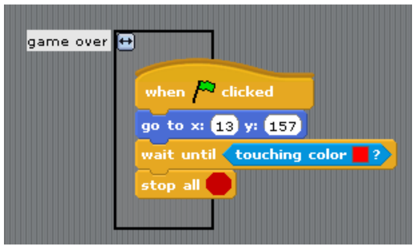
1. Introduction

The process of authoring a program can be described as going from ambiguous and informal representations about purpose and approach (mostly contained in the minds of programmers) to unambiguous, highly structured representations of instructions (mostly contained in computers). Code reading involves the reverse process: relating structured representations to general ideas about what the program does and how. Current development environments provide great assistance when working with formal representations, through intelligent completion, refactoring, and debuggers, for example. But relating formal representations that computers can use with informal representations that programmers can use remains, for the most part, the sole responsibility of the programmer.

Perhaps the scarcity of tools for working with informal representations is in part because informal representations are ambiguous. 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 [Green 1989]. Instead, we suggest that programming environments could permit programmers to describe the desired program in more natural and informal terms, even if the terms themselves or the relations between them are ambiguous, then assist the programmer in the task of concretizing the structure and content of the program into an unambiguous executable form.

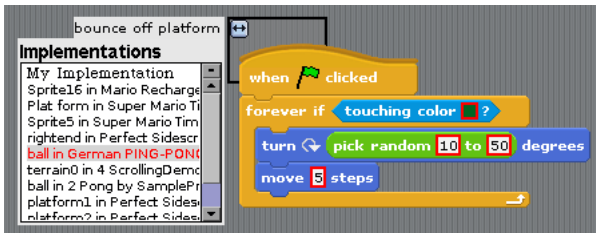
2. Frontend: Zones

In this paper, we present one initial example of a system that can work with informal representations of software. Our interface, called *Zones*, uses informal statements of the general purpose of code fragments to help programmers find and share reusable code. A *Zone* is an active association between a code fragment and a brief natural language statement of its purpose—“What’s this for?”—not unlike a comment. Programmers can describe the purpose however they think about it; the statement need not be precise, comprehensive, or even grammatical. By only giving one line, the Zones interface encourages purpose statements to be short. Figure 1 shows



```
when [flag] clicked
go to x: 13 y: 157
wait until <touching color [ ]?>
stop all
```

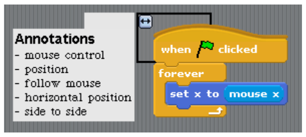
Figure 1. A Zone (black rectangle with flag) associates a fragment of Scratch code with a statement of its purpose. The top-left button invokes the search sidebar.



My Implementation
Sprite16 in Mario Recharge
Plat form in Super Mario Ti
Sprite5 in Super Mario Tim
rightend in Perfect Sidescr
ball in German PING-PONG
terrain0 in 4 ScrollingDemo
ball in 2 Pong by SamplePr
platform1 in Perfect Sides
platform2 in Perfect Sides

```
when [flag] clicked
forever if <touching color [ ]?>
    turn (pick random 10 to 50) degrees
    move 5 steps
```

Figure 2. Given a purpose statement, the Zone sidebar (left) shows code that might fulfill it. Selecting an implementation from the list on the left shows its code on the right. As a simulation of future functionality, red boxes surround values that vary among otherwise similar code, highlighting what might need to be changed.



- mouse control
- position
- follow mouse
- horizontal position
- side to side

```
when [flag] clicked
forever
    set x to (mouse x)
```

Figure 3. The Zone sidebar can suggest possible purpose statements for a code fragment (simulated for this illustration).

an example where a programmer has used a Zone to annotate a short fragment of Scratch code. Other purpose statements from our users, who were developing video games, include "stay on path" and "Bounce ball around room."

Omitting either the purpose statement or the code turns annotation into search. If you provide an English statement of purpose and click the search button, the system searches for code that accomplishes that purpose (see Figure 2). Alternatively, you can mark some code and then search, which means: find how other people have generally described the purpose of code like this (see Figure 3). The search results are presented as a floating sidebar. The code for selected results appears within the Zone. You can then immediately try running the program with the transplanted code, make modifications first, or simply scroll through the results to get a general sense of different approaches to your task. Selecting the zeroth result returns to whatever code was in the Zone before, if any. A second click on the search button collapses the sidebar, leaving behind only the code and the search query, which then functions as an annotation, using exactly the language you chose when looking for the code to begin with. In this way, search queries seamlessly become annotations; though the code may require further tweaking, the annotation remains unless the programmer explicitly removes it. When the project is shared, that pair of annotation and code fragment is added to the database, either as new information or as a new explanation for existing code. Or, if in fact none of the search results were relevant (likely because the corpus of annotations is still small), code built within the bounds of that annotation becomes the first example of how to do that. The next programmer to look for related code fragments or purposes will benefit from the added annotated code. It could be said that by capturing both successful and unsuccessful search interactions, Zones learns how programmers describe purpose.

2.1 Scratch Programming Envorinment

While informal software representations are 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 and development environment 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 events, including user input and named messages from other sprites. The language includes control flow statements, Boolean and mathematical expressions, and sprite- or project-scoped variables. Though the detailed techniques of this paper are somewhat tailored to Scratch, the general approach should be adaptable to other languages.

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 more reusable *a priori*, despite the general lack of software engineering discipline in the programmer community. Also, while the language lacks procedures or functions in the traditional sense, the event handler design makes concurrent modularity idiomatic and greatly simplifies task coordination, which tends to make a large amount of code reusable without significant modification. 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.

2.2 Other Types of Interactions with Informal Software Representations

The purpose-driven code reuse of Zones is one of many ways that a development environment capable of working with informal software representations could help programmers. For example, a subgoal may be inferred from a higher-level goal and the code written so far by making analogies to the subgoals of other similar projects. Similar techniques could apply to other artifacts as well: for example, a goal could map 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.

Second, insomuch as failed test cases or other bugs indicate a failure of the concrete code to satisfy the programmer's goal, we can also think of debugging in the context of informal representations. For instance, we could collect examples of solutions to particular problems in particular situations, then analyze and generalize them to learn what problems might come up in a certain situation and what types of code changes tend to fix them. The result would enable debugging by making 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. An informal survey of open-source code repositories suggests that a large amount of code has similar goals and subgoals, but it was not reused, due perhaps to differences in environment, constraints, or libraries, or simply being unable or unwilling to find and incorporate that other code. A development environment that can recognize the similarity of the programmer's intent to code already written, especially before the programmer has committed to a structured encoding of the task at hand, might encourage and facilitate code reuse. And by additionally capturing the process of refinement of high-level goals and maintaining links between the subgoal tree and the code, the development environment would be better equipped to help the programmer adapt that code to different scenarios or environments, even those that may come up at runtime.

3. Backend: ProcedureSpace

The Zones interface presents a unique set of requirements for its backend, called *ProcedureSpace*. Purpose statements could be treated as code search queries, but they would include abstract characteristics that would not match a keyword in an identifier, type, or comment (if present). And though we assume that some code fragments in the corpus have been annotated with one or more purpose statements, those annotations often differ in word choice and level of detail from the purpose statement that is queried for, and few code fragments are annotated. Finally, Zones searches may also go in reverse: from code to possible purpose statements.

These challenges preclude a straightforward or even multistage information retrieval approach. And since we allow purpose statements to be informal and ambiguous, formal reasoning techniques are particularly ill-suited. ProcedureSpace approaches this challenge with an unconventional technique: it uses the sparse and imprecise relationships between several different kinds of information to place them all in the same vector space (hence the name), where search queries become simple vector operations. This technique, called Blending [Havasi et al. 2009], is similar to standard information retrieval techniques, but uniquely illustrates a simple way of using multiple types of imprecise data together. The result is a representation that unifies syntactic knowledge about programs with semantic knowledge about goals.

Broadly speaking, ProcedureSpace aims to relate incomplete knowledge about two types of entities: purpose statements and code fragments. Purpose statements are handled with simple natural language processing techniques; code fragments are represented by characteristics from static analysis. The knowledge about purpose statements is incomplete both because they are themselves ambiguous and because the system's understanding of natural language is limited. And knowledge about code fragments is incomplete in that it is generally not known what characteristics of the code are relevant to accomplishing the goal and what parts are merely implementation details. However, associations between code fragments and purpose statements disambiguate each other: ProcedureSpace, in effect, learns about purpose statements from the code they describe, and learns about code by studying how people describe its purpose. ProcedureSpace additionally uses semantic background knowledge about English words and phrases to help understand the natural language statements.

Figure 4 shows an overview of ProcedureSpace's reasoning: it 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 characteristics of the structure of that code. This paper does not seek to establish that these particular kinds of data are necessary and sufficient for a natural language code reuse system, but rather to show how to relate these disparate kinds of data.

4. ProcedureSpace Implementation

ProcedureSpace uses the Blending technique of [Havasi et al. 2009] to relate English words and phrases, English purpose statements, code, structural characteristics of that code, and background knowledge ("features") about words. Blending works by constructing a matrix containing aligned data of several types, then using an approximate matrix factorization to represent each element (e.g., a code fragment or a purpose statement) as a vector. Relating elements to each other then reduces to simple vector operations, even for elements of

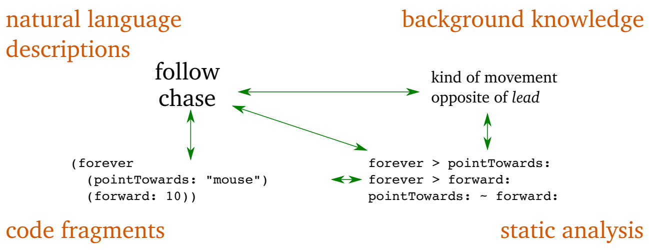


Figure 4. ProcedureSpace relates informal representations of purpose, such as the words "follow" or "chase", with code fragments and their characteristics.

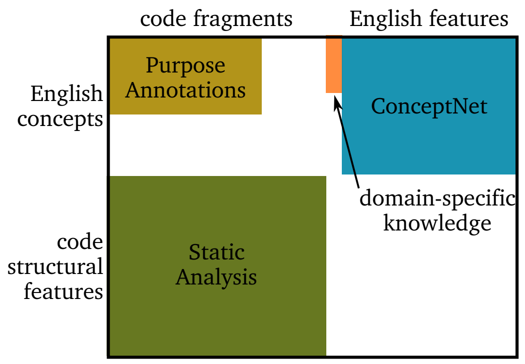
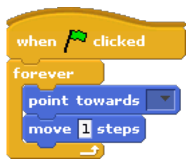


Figure 5. A simplified diagram of the ProcedureSpace analysis matrix, illustrating how purpose annotations bridge code static analysis with natural language background knowledge.



(a) as presented in the Scratch UI

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

(b) internal S-expression format

Figure 6. Example "chase" code fragment, taken from the Scratch project "Enemy AI tutorial: Chase."

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

Table 1. Selected structural features for the example code fragment

different types. Figure 5 shows a simplified diagram of the matrix, illustrating how purpose annotations bridge code static analysis with natural-language background knowledge. This section describes how each sub-matrix is constructed, then shows how to combine them to enable the queries that the Zones front-end requires.

4.1 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 constraints about types (e.g., "returns an integer") or interactions (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, either as a parameter or as the body of a conditional or looping construct

Containment A code element is contained within another code element (generalizes "Child")

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 feature types, 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 6, which makes a sprite chase or follow another sprite. Table 1 shows examples of the code structural features extracted for the example code. We 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 $CS(\textit{feature}, \textit{fragment})$ is 1 if *fragment* has *feature*, 0 otherwise. To keep long code fragments with large numbers of features from having a disproportionate effect, we normalize all code fragments to have unit Euclidean norm. Figure 7 shows a sample of the final matrix CS.

CS =

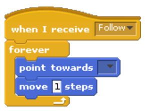
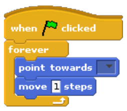
			⋯
Clump `[forward_ pointTowards_]`	0.27	0.27	⋯
Presence `EventHatMorph`	0.11	0	⋯
⋮	⋮	⋮	⋱

Figure 7. A sample of the code static analysis matrix CS

4.2 Annotations

Each Zone is an annotation that links a code fragment with a statement of purpose. These annotations are collected into a matrix: $AD(\textit{purpose}, \textit{fragment})$ counts the number of times that *fragment* was annotated with *purpose*. The purposes are actually stored as (`Purpose,` *purpose*) tuples to distinguish complete annotations from words that will later be extracted from them.

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, for a total of 100 annotations at the time of these experiments.

The matrix AD only accounts for purpose statements that are equivalent with respect to string equality. To allow inexact matches of annotations and search queries, we construct a matrix AW that relates code fragments with words and phrases extracted from their purpose statements using standard natural language processing techniques (e.g., lemmatization and stopword removal). Since similar annotations yield similar words and phrases, they will be counted as more associated.

To account for the sparsity of annotations and comments, ProcedureSpace also extracts tokens from identifiers: names of variables and events (analogous to function or method names). Token extraction additionally includings splitting underscore_joined and camelCased strings. Each element of the resulting word-code matrix, $WC(\textit{token}, \textit{fragment})$, counts the number of occurrences of *token* in *fragment*..

4.3 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 indicate how English concepts are similar to each other. Though many other lexical resources are available, ConceptNet uniquely provides a broad coverage of goals and actions, in a plain-language form that is amenable to imprecise reasoning and usage in a user interface.

4.3.1 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 [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.

4.4 Blending

Given the six matrices of relationships, we then combine them using Blending. To apply the Blending technique to a set of matrices D_i, align the labels (filling in zeros for missing entries), then add the matrices together:

$$A = \sum_i \alpha_i D_i$$

where α_i is a weighting factor that is adjusted to maximize the interaction between the datasets. (For these experiments the weighting factors were set manually, though [Havasi et al. 2009] presents an automated technique.) 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 the highest-variance dimensions of the data. The entries along the diagonal of Σ weight each axis, roughly indicating its importance in describing the patterns of relationships among the items of the matrix.

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. CS relates code fragments to code structural features, while background knowledge relates English concepts to English features. The

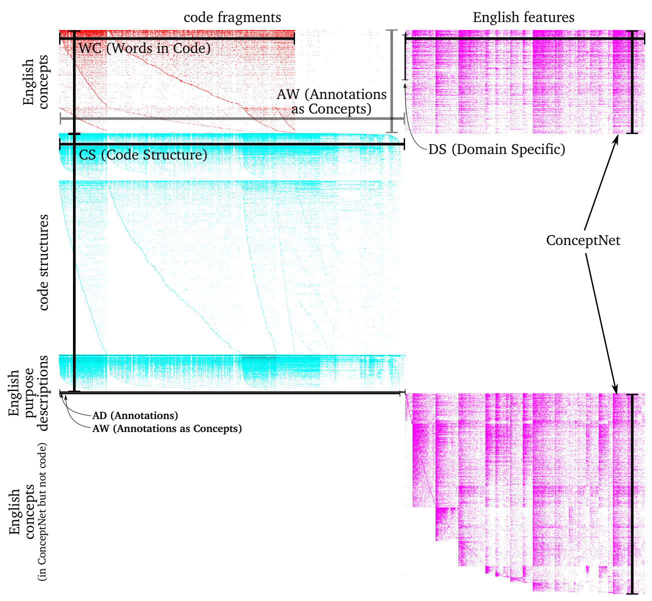


Figure 8. Layout of the ProcedureSpace matrix in detail, showing the origin of each element

annotations link the two disparate domains by relating code fragments to English concepts. Figure 8 details the layout of the ProcedureSpace matrix, showing where each entry in the matrix comes from. Effectively, the purpose annotations bridge the structural features derived from static analysis with the natural language background knowledge in ConceptNet.

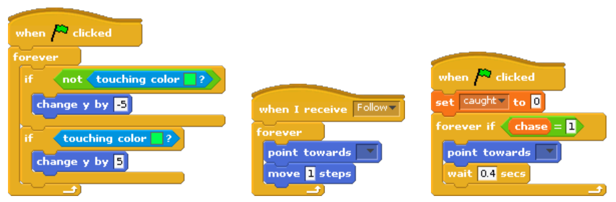
4.5 Goal-Oriented Search

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 purpose statement composed of English words w_i, you simply sum the corresponding vectors:

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

where the : notation indicates a slice of an entire row. Then to find how well that statement may apply to a particular code fragment, you take the dot product of $\vec{p}$ and that code 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 $\vec{q} = \sum_{i=0}^{n} U\left[f_i, :\right]$, 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 statement with the code fragments in that project.



```
when [flag] clicked
forever
    if <not <touching color [■]?>>
        change y by (-5)
    if <touching color [■]?>
        change y by (5)
```

```
when I receive [Follow]
forever
    point towards [ ]
    move (1) steps
```

```
when [flag] clicked
set [caught] to (0)
forever if <(chase) = (1)>
    point towards [ ]
    wait (0.4) secs
```

(a) "gravity" (b) "follow player"

Figure 9. Sample search results

4.6 Search Results

Users of our system searched for a variety of goals and expressed them in a variety of ways. Figure 9 shows selected results for some queries that users performed. The first search, "*gravity*," exactly matched the annotation of a code fragment, so it was returned as a search result. This result illustrates that 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 6 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 the behavior of 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.

5. Preliminary User Experience

"Searching by goal is a really different way of programming," said one participant in our preliminary user study. We sought to understand whether the Zones interface (both concept and implementation) helps programmers make and use connections between natural language and programming language. 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.

After a sample task to familiarize the users with interacting with the Zones interface, they were presented with a project containing many sprites, each exhibiting some readily observable behavior, and were instructed to duplicate the behavior of one or two sprites, using Zones if they wanted. They were not given cues for how to describe that behavior so that their queries would be as natural as possible. 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. One 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 encouraged 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.

6. Related Work

6.1 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.

6.2 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 knows what keywords should appear in a line of code but not exactly how to form that line of code, 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 already be precise enough to know almost exactly what should happen at each line. 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 a natural language dialog paradigm, it could be very helpful for interactions where relating informal representations to code requires more than a single step.

6.3 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 approximate inference used by our system trades guarantees of correctness for the ability to make plausible inferences efficiently across a wide variety of programming situations.

7. Conclusion

We suggest that programming environments should be able to help programmers work with the informal representations that are a natural part of the programming process. One particular difficulty in working with informal representations of software is that they can be ambiguous. We suggest that given today's large-scale code repositories, programming environments can, instead of shunning ambiguity, manage it by relating ambiguous statements to concrete examples.

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 when computers can work with programs both in natural language descriptions and in code, they can come closer to really understanding what we want them 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.

T. R. G. Green. Cognitive dimensions of notations. In *Proceedings of the fifth conference of the British Computer Society, Human-Computer Interaction Specialist Group on People and computers V*, pages 443–460, New York, NY, USA, 1989. Cambridge University Press. ISBN 0-521-38430-3.

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.