Vajra: Step-by-step Programming with Natural Language

Viktor Schlegel
University of Manchester
Manchester, United Kingdom
viktor.schlegel@manchester.ac.uk

Benedikt Lang
University of Passau
Passau, Germany
mail@blang.io

Siegfried Handschuh
University of Passau
Passau, Germany
siegfried.handschuh@uni-passau.de

André Freitas
University of Manchester
Manchester, United Kingdom
andre.freitas@manchester.ac.uk

ABSTRACT

Building natural language programming systems that are geared towards end-users requires the abstraction of formalisms inherently introduced by programming languages, capturing the intent of natural language inputs and mapping it to existing programming language constructs.

We present a novel end-user programming paradigm for Python, which maps natural language commands into Python code. The proposed semantic parsing model aims to reduce the barriers for producing well-formed code (syntactic gap) and for exploring third-party APIs (lexico-semantic gap). The proposed method was implemented in a supporting system and evaluated in a usability study involving programmers as well as non-programmers. The results show that both groups are able to produce code with or without prior programming experience.

CCS CONCEPTS

• **Human-centered computing** → **Natural language interfaces**; **User interface programming**; *Text input*; Usability testing.

KEYWORDS

interactive programming; end-user programming; semantic parsing; IDEs; novice programmers

ACM Reference Format:
Viktor Schlegel, Benedikt Lang, Siegfried Handschuh, and André Freitas. 2019. Vajra: Step-by-step Programming with Natural Language. In *24th International Conference on Intelligent User Interfaces (IUI '19), March 17–20, 2019, Marina del Ray, CA, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3301275.3302267

1 INTRODUCTION

Natural Language Processing (NLP) focuses on the creation of methods and tools to automate the interpretation of natural language. Semantic parsing is one particular field within NLP which focuses on the mapping of a natural language sentence into a formal, i.e. machine-understandable representation of its meaning [8].

One application of semantic parsing is the development of end-user programming (EuP) models using natural language [17, 30]. Systems that can be programmed without the knowledge of a formal programming language could elevate programming tasks to the possibly inexperienced end-user. End-users can be experts from domains where programming tasks needs to be carried out on a regular basis but learning a programming language is not a core part of their education. A typical example is a psychologist that needs to analyze data gathered from experiments. An end-user programming tool can free him from the burden of learning the syntax of a programming language and the large set of functions required to build a valid program.

However the understanding of natural language in the context of programming is a complex task: First, a system must be capable of interpreting a natural language statement, to efficiently identify possible atomic language units such as control flow statements, functions or parameters and how to combine them to best reflect the *intent*. Second, the input has to be mapped to those units *semantically*, that is by their meaning rather than by mere syntax, as the end-user is potentially unaware of the concrete nomenclature used to refer to those units. This problem is known as *the vocabulary problem* [9].

Consider the following example: A user is asked to implement a greeting routine for an assistant robot in a hospital environment - the robot should issue a greeting, then turn left and move out of the way. The robot programming interface (further called *knowledge base*) provides the possibility to issue different high level commands to perform actions, such as moving, speaking or grabbing an object.

Our investigations, analogous to a study about end-user programming learning barriers by Ko et al. [13], rephrase the problems the user is confronted with in order to successfully implement such a routine as the following questions:

(1) How is the knowledge base of actions organized, i.e. what is its syntactic structure (e.g. return and parameter types)?
(2) What are the required actions and their names?
(3) How to compose those actions to express a concrete routine?

Current approaches like Choreographe, [28] Ruru [5], VPL [21], and modern IDEs offer assistance to answering questions (1) either by visualizing the program or providing convenience functions like indexing the knowledge base and syntactic code completion, they rarely focus on the question (2) and other than template and

IUI '19, March 17–20, 2019, Marina del Ray, CA, USA

ACM ISBN 978-1-4503-6272-6/19/03…$15.00
https://doi.org/10.1145/3301275.3302267

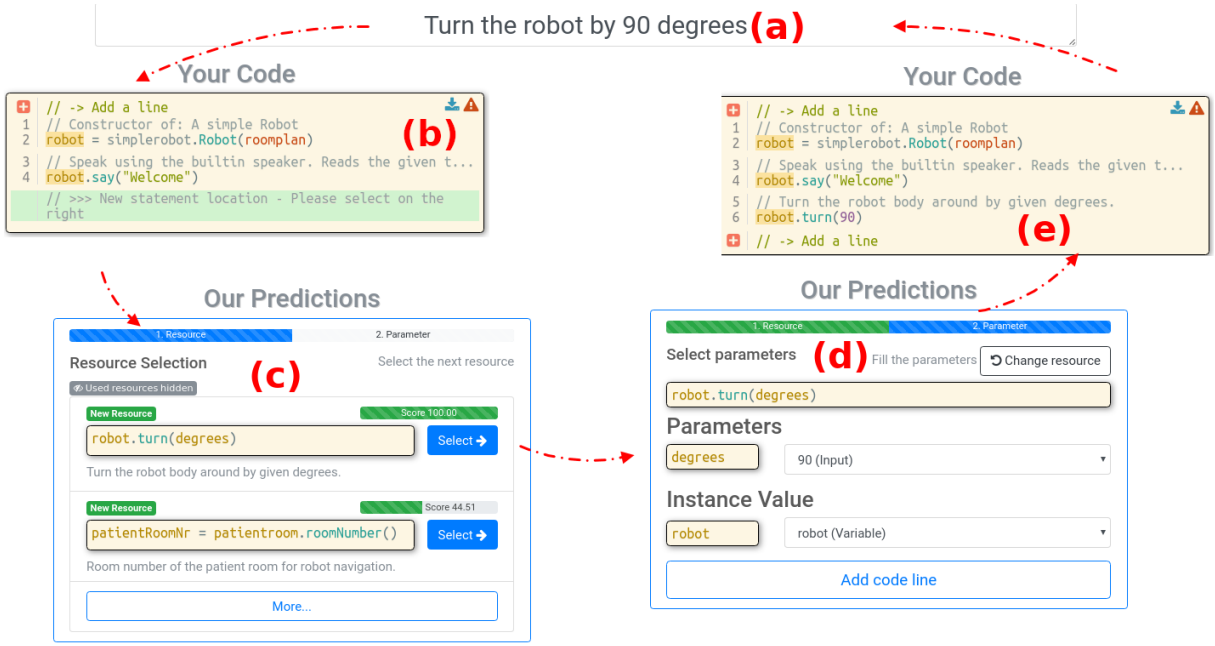

Figure 1: Vajra in action: The user issues a natural language command (a) and chooses its position in the existing code (b). The system generates a list of possible statements (c) and their associated parameters (d), semantically most similar to the input. Upon user choice (disambiguation) the concrete statement is inserted into the code (e) and the user can proceed to the next statement (a).

boiler-plate code, they provide little help to capture the users intent in source code (3). Even if the end-user has figured out how to express his ideas in the formal way the programming language requires him, he still has to match his vocabulary to the one of the knowledge base. On the other hand, approaches that convert natural language to source code directly [4, 10] are capable of capturing the intent of natural language by learning natural language patterns from a big training set. Further they are capable of bridging the gap between the API vocabulary and the user vocabulary by injecting background knowledge about word similarity, albeit indirectly, by using external word representations that are pre-trained on large-scale textual corpora [23, 27]. However these approaches sacrifice transparency and control, as the output of such a model is impossible to interpret with regards to validity. In addition, due to their dependency on deep learning, their performance is dependant on a high-quality training set.

To achieve the best from both worlds we introduce a hybrid approach and the end-user programming platform Vajra that implements it. Its core features are:

- **Iterative Workflow:** Users create their program statement by statement. This design choice helps to address question (3). We argue that this way, the user is able to discover the solution to a problem in a guided way and directly controls the result, in contrast to a fully-automated translation from input to a complete program.
- **Natural Language Support:** By parsing natural language input and *semantically* matching it with existing resources, resulting in well-formed program statements, we address questions (1) and (2). We argue that this allows end-users to create programming statements independently of the vocabulary of the knowledge base.
- **Live Program Analysis:** The statements generated by the natural language parser are further refined by the symbolic dependencies of the existing code from previous steps.

Figure 1 shows a usage scenario:

(a) The user issues a written utterance in natural language using his vocabulary that will be translated into a programming language statement.
(b) He choses the position in the code editor where the statement should be inserted.
(c) From a list of proposed statements that are semantically most similar to the input the user choses one that fits best.
(d) The user sets the statement's parameters that are automatically parsed from the input or deduced from existing code.
(e) Finally the statement with its chosen parameters is inserted into the code and the user can proceed with the creation of the program.

To our knowledge Vajra is the first attempt to combine a vocabulary independent semantic parsing approach with an iterative

workflow to create an environment for domain experts and casual users to compose programs via natural language. To evaluate whether this approach can help end-users with their programming tasks, we conducted a usability study with 12 participants with differing experience in programming. They were asked to implement various routines for a robot in a hospital setting. While doing so, their time to complete the tasks was measured and they were asked to fill out a questionnaire. We found that participants were able to successfully finish the tasks and were confident to use the system only after a short introduction.

The contributions of this paper are:

- The idea of supporting users by allowing them to incrementally create a program by parsing natural language into new statements based on the existing program in a *vocabulary-independent* fashion;
- The underlying semantic parsing model, which utilizes a combination of a dependency analysis, distributed semantics and a ranking algorithm to provide an adaptable, transparent and human-interpretable prediction model;
- Vajra, a prototype IDE implementing this idea;
- A usability study, which shows that Vajra can be used by programmers as well as non-programmers to efficiently create programs.

2 BACKGROUND AND RELATED WORK

The term end-user programming is defined as: "programming to achieve the result of a program primarily for personal, rather public use." [12, Ch. 2.2]. Vajra is categorized as an end-user programming platform, a platform that allows end-users with and without programming experience to build an application [31, p. 5]. Since our system is designed for building a program instruction by instruction, a certain level of understanding of programming concepts is needed. This requirement categorizes our proposed system as an end-user professional programming platform [31, p. 5], but as the evaluation shows, end-users can easily reach this level of understanding about programming concepts after using the platform for a short time.

Template Code Generation. An approach of combining natural language processing with end-user programming was taken by Metafor (2005) [19]: In Metafor, the user tells a natural language story. The system is processing this story and produces Python source code in the form of a class skeleton with partially implemented methods. Based on a common sense knowledge base, the system is inferring which objects of the natural language story should be translated into methods and classes. For example, based on the story "There is a bar with a bartender who makes drinks.", Metafor would produce two nested classes 'bar' and 'bartender' with a single method 'make(drink)' [19]. Based on existing background knowlege, Metafor is inferring that the bartender is an actor and should be presented as a class with a method "make(drink)". Instead of producing a finished application, Metafor is intended for brainstorming [19, Ch. 4]. While Vajra works on a predefined knowledge base, which defines available operations and its documentation, Metafor creates code which could act as a knowledge base for Vajra, instead of using existing operations of the knowledge base. NLP for NLP (natural language processing for natural language programming) [22] is another approach combining natural language processing with end-user programming. This approach generates low-level code, specialized in generating loops and the corresponding steps. For the user query "Write a program to generate 1000 numbers" it generates a 'for' loop of 1000 iterations and adds a step "generateNumber(i)". Other work focused on translating natural language to high-level code or in the case of [1] to an object-oriented model based on UML, which also can be used to generate code skeletons [1]. Instead of using a knowledge base of code operations, the field of human-robot interaction provides similar needs but mapping natural language to a set of robot interactions.

Semantic Parsing. The work of Lauria et al. [14] covers the programming of robots using natural language. Based on speech, transferred to natural language and initially processed using a syntactic parser, a robot is controlled by executable commands for robot navigation defined by a knowledge base. A similar approach was chosen by Eppe et al. [6] based on deep semantic reasoning and robot-human dialog for disambiguation. These approaches are very similar to ours because the natural language is processed by semantic annotation and mapped to robot functions which form a program. While our approach allows the user to compose instructions in any order, these systems focus on sequences of navigation commands and their underlying state transitions.

Search Based. Modern IDE's support programmers with string-based API search after an initial indexing step [7, 11, 20]. Tools like CodeBroker [32] or PRIME [24] take the current code state into account to provide contextualized search results. Little et al. [18] use a keyword-oriented search to overcome the vocabulary gap. Their approach is different to ours, as it does not involve a semantic parsing step and is thus not capable of accepting actual sentences as input.

Program Synthesis. The idea of composing programs automatically borrows concepts from the rich literature of program synthesis. Modern approaches in this branch try to to code approaches aim to utilize deep learning models to convert natural language into a set of operations directly [4, 10]. However, in order for a user to understand the program, and specifically to verify the assumptions about the API he uses, we propose to use an interactive method of code creation instead. In this way, users can see how and why different natural language commands result in different statements and adapt to it, as opposed to a black-box approach which will either result in a desired program or a spurious one, without providing an explanation. CodeMend [29] is very similar to our approach as it translates natural language commands into code or modifies existing code into natural language, taking into account the current code state. However they scrap the web for existing code to train their prediction model and to predict results. Thus CodeMend is not capable of dealing with APIs not seen during training time without adapting the whole model to take this API into account. While our approach also needs to be trained on new APIs, the training set is comparatively small as we don't utilize a complex machine learning architecture and can easily be automatically constructed from the API documentation. In conclusion, CodeMend will excel with popular APIs such as *scikit-learn* [26] due to the abundance of

example code on code sharing platforms like GitHub[1] and StackOverlow[2], while our approach is geared towards APIs with scarce to no resources available.

3 END USER PROGRAMMING PLATFORM

Vajra assists the user in building a program step by step. Individual instructions are added by the user from a list of predictions, which are based on the user's query as well as other surrounding statements. The iterative workflow to create a program is illustrated in Figure 2 and can be summarized in 5 simple steps:

(1) Enter a query as a natural language command, to express what the program is intended to do next
(2) Select a location for the next statement (in most cases it is the natural sequence)
(3) Select a statement from the list of proposed statements
(4) Select proposed values as parameters of the statement
(5) Add the statement to the program

The user can change the query at any time to update the prediction list.

3.1 Language model

The program composed by the user is a list of statements which utilize a sub-set of the Python language with the following restrictions:

- variables names must not be re-assigned.
- only function calls and object oriented composition (dot (`.`) operator) are allowed. This means that control-flow elements like `for`, `if` or `while` and convenience functions like `lambda` expressions or list comprehensions are not among the proposed statements.

While it may seem as a limiting factor at first, the control flow elements can be modeled by the knowledge base to support them (with functions like `action.executeIf(condition)`). We argue that these restrictions facilitate the understanding of the program and ensure that the user can only build upon resources designed and documented by the developer of the knowledge base.

3.2 Semantic Parsing

To provide a *semantic approximation* prediction system based on the userâĂŹs query, this query must first be processed and the semantic information extracted. The process of extracting the semantic meaning of a sentence is called *semantic parsing*. In a first step, the userâĂŹs query is annotated to mark sections extracted in later steps. This annotation procedure is divided into multiple processing steps, each building upon the previous one to further refine the annotation and is shown in Figure 3.

1 The syntactic parser tokenizes the sentence and extracts Part-of-Speech (POS) tags as well as the dependency structure.
2 The *Semantic Type Tagger* detects special information like file names or numbers in the natural language input. We deploy a rule-based approach to detect those. This enables a precise parameter proposal generation.
3 The *Reported Speech Tagger* identifies reported speech inside the input which is important since it might be used by the user to communicate (string) values as parameters to the system.
4 The *Lexical Role Labeler* pre-processes the annotated tokens and assigns them their lexical role (such as noun, verb, verb objects). Furthermore it detects stop words.
5 As a last annotation step, the *Semantic Role Labeler* is responsible for labeling the lexical role objects by their semantic role in the context of predicting resources and parameters. It's possible outcome can be seen in Table 1. It uses all annotation information from previous steps to create a semantically annotated and filtered tree structure of semantic roles and relationships between them. This tree, in contrast to the dependency tree, is based on parts of the sentence instead of individual tokens.

The result of the annotation is visualized in Figure 4.

To compare the userâĂŹs input with items from the knowledge base, only semantically relevant parts are extracted from the output of the Semantic Role Labeler and collected as a tuple of candidates (M, P_n, P_v, P_{nv}) where

- $m \in M$ represent the user's main intent in the form of objects and actions
- $pn \in P_n$ are possible parameter names.
- $pv \in P_v$ are possible parameter values
- $pnv \in P_{nv}$ is a list of tuples representing possible detected mappings between parameter names and values

We use a rule based approach illustrated in Algorithm 1 to generate the candidate lists from the annotated semantic roles and their relationships. `handleDisconnectedPairs` in this context is a function that detects additional mappings *pnv* that were not detected by the Semantic Role Labeler, such as $(degrees, 90)$ in the input sentence "set degrees to 90". The extraction output of our guiding example is seen in Figure 4.

3.3 Statement proposal generation

Based on the processed input a list of predicted resources, which can be selected as new statements, is presented to the user. To predict these resources, additionally to program analysis, the system uses semantic approximation between the resources of the knowledge base and the user's query. These resources must then be ranked based on a feature vector to form a sorted list of predictions. To form f_s, for every knowledge base resource and the processed input,

Table 1: Possible semantic roles and relations between them.

Role	Description
`ACTION`	Verbs. describe interactions between objects
`OBJ`	subjects and objects. represent an entity associated with an action
`VALUE`	semantic types and reported speech. candidates for primitive type parameters
`MOD`	Modifiers such as adjectives. Define further attributes of objects and actions
Relation	**Description**
`MOD`	Modifier relations (e.g. 'green robot')
`ATTR`	relations between `VALUE` and `OBJ` objects

[1] https://github.com
[2] https://stackoverflow.com

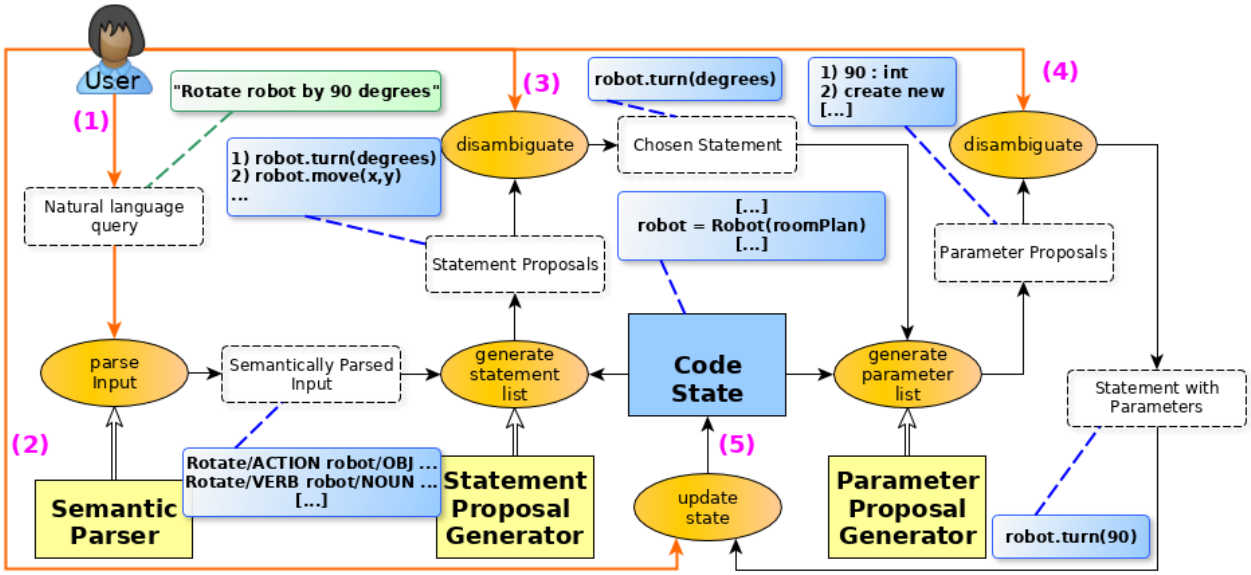

Figure 2: Iterative workflow (User interactions are in orange): (1) the user issues a query and (2) chooses where to insert the result in the code editor. (3) he choses one statement from a list of proposals and (4) choses its corresponding parameters. (5) the statement is inserted into the editor.

the sets of pairs $M_{name}, M_{desc}, P_{name-name}, P_{name-desc} P_{value}$ are built where

- M_{name} is the Cartesian product of M and the resource name
- M_{desc} is the Cartesian product of M and the resource documentation
- $P_{name-name}$ is the Cartesian product of P_n and the parameters of the resource
- $P_{name-desc}$ is the Cartesian product of P_n and the documentations of the resource parameters

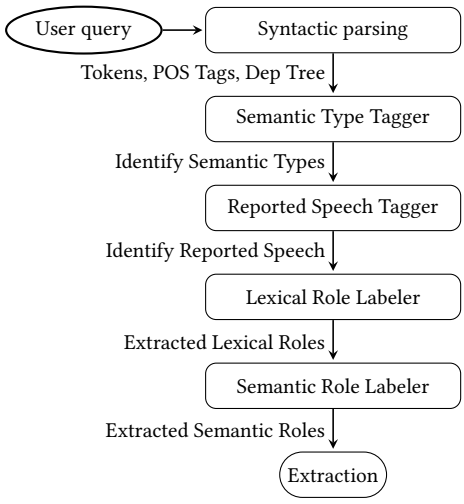

Figure 3: Pipeline of processing steps to annotate the user query

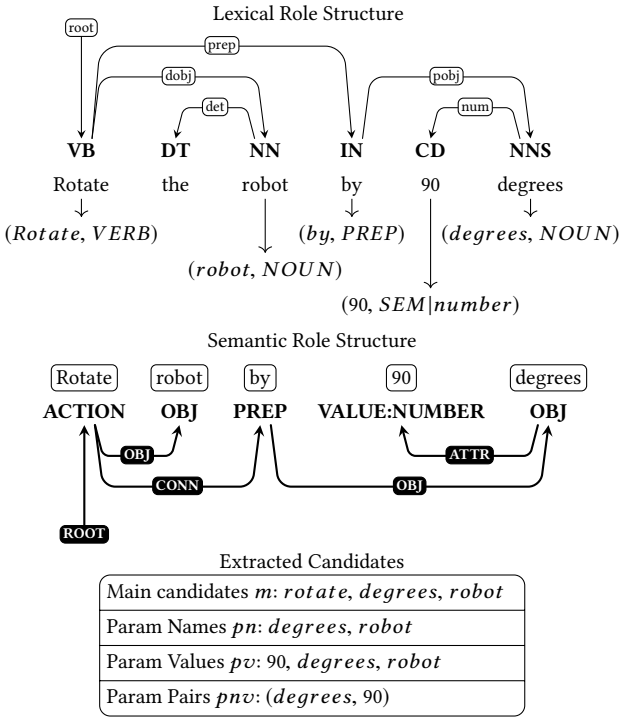

Figure 4: Semantic Parser output for the phrase "rotate the robot by 90 degrees"

- P_{value} is the Cartesian product of $P_{nv} \cup P_v$ and the documentations of the resource parameters.

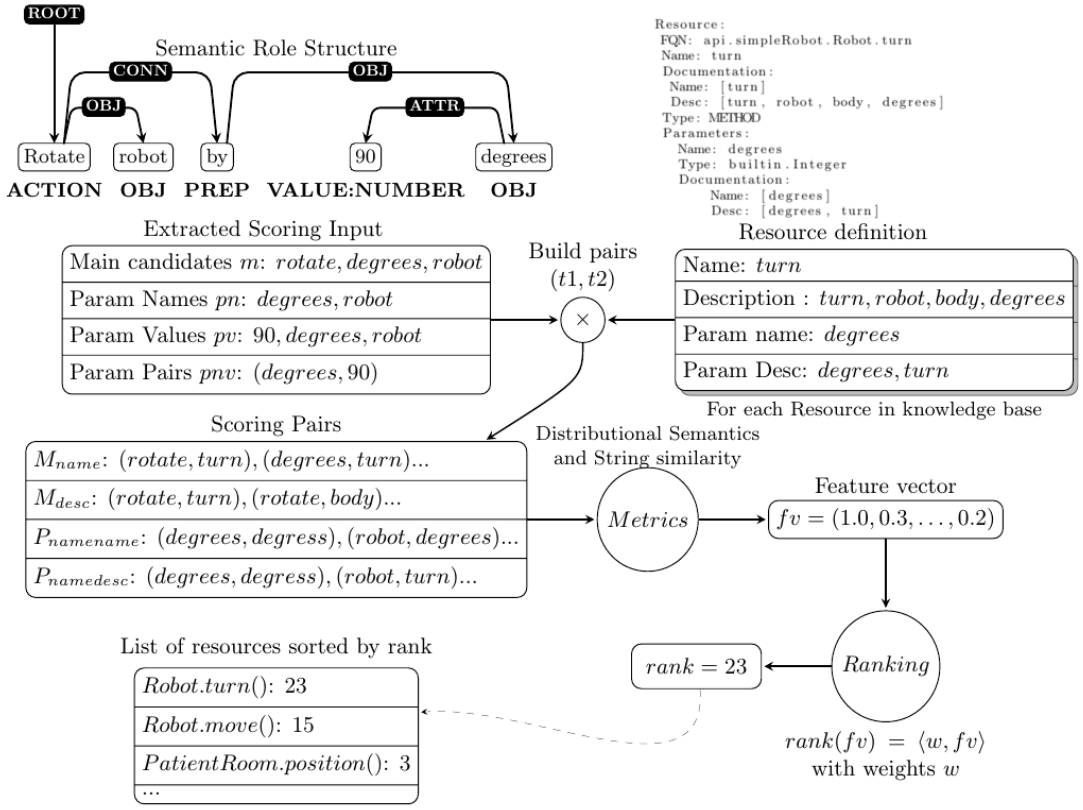

Figure 5: Statement proposal generation for the phrase "Rotate robot by 90 degrees" (higher rank means better matching).

The feature vector $f_s = (d_1, .., d_5, s_1, .., s_5)$ is formed according to Table 2. The resource list is ranked and enriched by a program analysis component which adds most plausible statements based on the current state of the code and the position of the statement insertion (such as initializing a variable that is used at a later point in the code or reusing an already initialized one). Note that the program analysis only outputs valid proposals, i.e. it will not propose to initialize a variable that is already initialized.

3.4 Argument proposal generation

Once the user has chosen a statement from the proposal list, he has to chose its parameters from the generated proposal lists or introduce a new variable, which needs to be initialized in the code before the inserted statement at a later time. In case the parameters types are primitives such as `int`, the list of proposals is generated from the parsed input. Therefore for every item in the knowledge base the sets of pairs $A_{name-name}, A_{name-desc}, A_{value-desc}$ are built, where

- $A_{name-name}$ is a Cartesian product of all pn and the name of the parameter
- $A_{name-desc}$ is a Cartesian product of all pn and the documentation of the parameter
- $A_{value-desc}$ is a Cartesian product of all pv and the documentation of the parameter

and $pn \in P_n, pv \in P_v$ if no mapping between possible parameter and values was detected in the parsed input. If a possible mapping was detected, the sets are built such that $(pn, pv) \in P_{nv}$ accordingly. Note that this is reflected in the ranking vector representation, to weigh inputs with an explicit mapping accordingly. The ranking vector $f_p = (d_1, ..d_3, s_1, ..s_3, i)$ is formed according to Table 2.

Table 2: Feature vector components used for ranking the statement and parameter proposals. δ is a similarity function, based on either string similarity (s) or distributional semantics (d), depending of the vector component.

Comp	Description
	Statement proposal feature vector
d_1, s_1	*max* of $\delta((t_1, t_2))$ for all $(t_1, t_2) \in M_{name}$
d_2, s_2	*max* of $\delta((t_1, t_2))$ for all $(t_1, t_2) \in M_{desc}$
d_3, s_3	*max* of $\delta((t_1, t_2)))$ for all $(t_1, t_2) \in P_{name-name}$
d_4, s_4	*max* of $\delta((t_1, t_2)))$ for all $(t_1, t_2) \in P_{name-desc}$
d_5, s_5	*max* of $\delta((t_1, t_2)))$ for all $(t_1, t_2) \in P_{value}$
	Parameter proposal feature vector
d_1, s_1	*max* of $\delta((t_1, t_2))$ for all $(t_1, t_2) \in A_{name-name}$
d_2, s_2	*max* of $\delta((t_1, t_2))$ for all $(t_1, t_2) \in A_{name-desc}$
d_3, s_3	*max* of $\delta((t_1, t_2)))$ for all $(t_1, t_2) \in A_{value-desc}$
i	1 or 0, indicates detected parameter mappings

Algorithm 1 Rule-based Scoring Input Extractor

Input: Semantic role objects *semobjs*
Output: candidates $si \leftarrow (M, P_n, P_v, P_{nv})$

```
si ← (M, P_n, P_v, P_nv)
for node ∈ semobjs.nodes do
    if node.type == ACTION then
        si[M].add(node)
    else if node.type == OBJ then
        si[M].add(node)
        si[P_n].add(node)
        si[P_v].add(node, type(node))
    else if node.type == VALUE then
        si[P_v].add(node, type(node))
    else if node.type == MOD then
        si[M].add(node)
        si[P_v].add(node, type(node))
    end if
end for
for (lhs, edge, rhs) ∈ semobjs.edges do
    if edge == ATTR then
        si[P_nv].add((lhs, rhs, type(rhs)))
        si[P_n].add(lhs)
        si[P_v].add(rhs, type(rhs))
    end if
    if edge == MOD then
        si[P_v].add(rhs, type(rhs))
    end if
end for
handleDisconnectedPairs(semobjs)
return si
```

Again, the list of proposals generated from the input is enriched by valid proposals based on program analysis, such as already initialized variables.

3.5 Semantic Result Ranking

In general, the positions of the vector f_s and f_v represent the semantic similarity between different aspects of the parsed user input and parts of the knowledge base items. We use two similarity measures:

- **String distance based:** A string distance function such as Levenshtein distance [15] measures the *lexical difference* between two terms by counting the minimal number of required changes, deletions or additions of characters to match a pair of strings [25, p. 37].
- **Distributional semantics based:** Distributional semantic similarity measures the distance between two word embeddings and is capable of capturing the *semantic similarity* between two terms. We use a model based on word2vec [23] embeddings trained on a corpus of Wikipedia articles.

To perform the ranking based on the feature vectors we train a *ranking function* to assign weights to different features and rank accordingly. We use a *learn-to-rank* approach [16]. By training a *Support Vector Machine (SVM) classifier*, following the *pair-wise ranking* approach [16, Ch. 4.1]. The classifier has to be trained for each knowledge base separately. For our knowledge base used in the evaluation, comprising 13 classes and 30 resources definitions, the creation of a small dataset took less than one hour by the knowledge base developer.

The whole proposal generation and ranking process for statements is illustrated in Figure 5.

3.6 Limitations and Scope

The scope of Vajra was set as a end-user programming platform. As such, convenience constructs such as Python's list comprehensions mainly aimed towards experienced programmers are not supported nor are they required. One might argue that simple control flow constructs could be introduced in order to allow end-users to exert more control over the program. We follow the philosophy that the designer of the knowledge base should be left to decide which control flow elements he wants the end-user to utilize and thus exposing them in form of functions.

One inherent limitation of the approach is that it heavily relies on the documentation quality of the knowledge base designer. Poor documentation of parameters, return values and types or badly chosen resource names may impact the semantic matching capabilities.

Additionally, in its current state, the user interface provides limited feedback whether the composed program is working and whether it is doing so in the intended way. While generated statements are syntactically well-formed by design and users are notified about uninitialized variables, a future iteration of the design might include a better handling of runtime errors such as a live-programming component that executes the code while it is being programmed.

4 EVALUATION

To measure the quality of the proposed approach, an evaluation was conducted to answer the following questions:

(1) Are users able to learn to use the system in order to solve programming tasks?
(2) Is programming expertise reflected in the time it takes users to solve a task?
(3) How complex do the natural language commands need to be in order to express the users' intent?
(4) What is the users' perception of the system's usability?

4.1 Setting

To answer these questions, we conducted a usability study. We asked users to implement two typical tasks involving programming a robot inside a hospital. The robot can execute actions like grabbing objects, moving and reading text aloud. First, users are required to solve a simple task with only an introduction to the most basic concepts of the system. Then, after a detailed explanation, the user is requested to solve a more complex task.

For both tasks, the time to solve the is tracked for each participant. Solved in this case means the resulting program behaves in accordance to the task description. To measure the general usability users were asked to fill out a questionnaire. The questionnaire is based on the *System Usability Scale* (SUS) [3], which consists of 10 statements. Users can agree with the statements to an extent by

giving scores on a Likert-type scale (where 1 is the least and 5 is the most).

In total, 12 volunteers participated in the evaluation. Five of them declared they are familiar with at least one programming language with a 4 or 5 on a Likert-type scale. Seven of them declared they have no prior experience in programming with a 1 and 2 on the scale.

Each participant was handed an evaluation briefing in paper form, consisting of a short explanation of the field of study followed by a basic description of the system's capabilities. The briefing contains two tasks for the user to complete the evaluation, which are specified below. These tasks describe a scenario the user needs to solve using the User Interface (UI). Each task is limited in time and monitored by a supervisor. The supervisor will create a test protocol containing relevant details (such as adding or removing a line of code or changing the query) and progress during the session. To start the evaluation, each participant opens the Vajra UI in the browser and tries to solve Task 1.

- **Task 1:** In the first task, the user is told to create a robot and let it say a predefined text. The time frame is limited to 10 minutes. The user is not presented with a full explanation of the system before this first task. Instead, the user is introduced to the task and presented with an overview of the three main control elements for orientation. The purpose of not introducing the user to all concepts before the first task is to evaluate the general usability and to get an unbiased impression on how users interact with an unknown prediction based interface.
- **Task 2:** All users start with the correct instructions needed to complete Task 1 and are asked to add further statements to complete the second task which requires the user to instruct the robot to grab an object and move to another room. This task is limited to 20 minutes and ends if the user completes the task or the time has run out.

This form of study gives insights in how users interact with the system, however it is limited in following aspects. Firstly, the number of participants is low, so the quantitative evaluation of the results is prone to noise. Therefore we focus on a qualitative analysis in the discussion. Secondly, we omit the evaluation of our prediction model in the traditional way, that is using gold-standards and their associated metrics. We chose to do so deliberately, because we're interested in how the user can benefit from our predictions rather than in performance of the model according to those metrics. Lastly, we are aware that our study design might be affected by an *experimenter bias*, where participants *(1)* might be influenced, albeit unintentionally, by the person conducting the study and *(2)* tend to give better grades out of sympathy. A future study carried out without human contact, for example using one of the existing crowd-sourcing platforms, will help combat this effect.

4.2 Overview

The evaluation showed that programmers, as well as non-programmers, learned the system quickly (26.7% improvement in time per instruction for non-programmers, a similar performance of both groups in the second task). All participants were able to solve the given tasks and rated the system with an "excellent" [2] usability score (average 85/100). Furthermore, they intuitively optimized their query length to a required minimum in both tasks (average of 2 words per query) to achieve the desired outcome.

The following sections discuss the users' *effectiveness* when using the system to perform the tasks, covering Questions 1 and 2, their *interaction* of with the system, covering Questions 3 and the *usability* of Vajra, covering Question 4.

4.3 Effectiveness and Efficiency

All participants were able to complete both tasks in the given time. To measure the users' efficiency during the tasks, the *average time per instruction* was measured for all participants. This measure reflects a user's efficiency to express his intent in code - lower time per instruction means the user is able to create desired programs more quickly. In the first task, without a guided introduction to the system, on average, the group of non-programmers took 27% longer per instruction than the experienced programmers (percentage change) as seen in Figure 6.

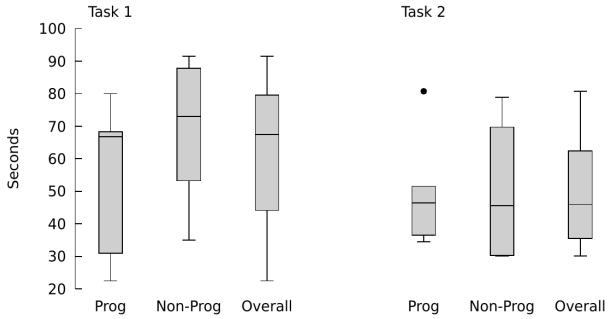

Figure 6: Time per instruction (seconds) on average per user for both tasks

After the guided introduction (Task 2), both groups achieved nearly the same performance, see Figure 6. We argue that this shows that the programming aspect of the system was encapsulated in a simple way, even non-programmers could grasp the concepts fast. In comparison to the performance of the first task, on average, the non-programmers improved by 26.7%, programmers by 7.0% (percentage change). This is consistent with the observation that non-programmers were much more trusting in the system than programmers, especially in the second task. Non-programmers often accepted the systems defaults and proceeded to the next instruction more quickly than programmers, after they understood the concepts. Another potential reason is that programmers tended to create variables first before referencing them, which counteracted the principles of the program analysis component of the system. The low variance (except for one outlier) in the group of programmers shows that programmers, despite being more careful, were more consistent in the use of the system. The marked out-lier in the programmers' group was the result of not trusting the system's prediction and based on confusion about a variable's name.

4.4 User interaction

The evaluation has shown that users have no difficulties using the system after a basic introduction. An important factor on how the participants were using the system is the length of the query: All participants, even users without a programming background, were using very short user queries $n <= 3$ for 76.5% of the time, according to Table 3, especially to query variable names and types directly. They intuitively optimized the query to a bare minimum such as 'robot move' or 'create flower', as soon as they realized that it is sufficient to get the right predictions. Since the method of program analysis predicts resources based on unbound variables, users realized they do not have to change their query every time to solve those dependencies and changed their query only to add statements which could not be solved otherwise. This supports the fact that complex syntactic structure is not required to express a command in natural language to be parsed by a semantic parsed nor does it seem desired by users. Hence optimizing on parsing long or nested sentences would not benefit the user in a system designed as ours, which gives important insight for future work.

Table 3: Number of words in the user query, grouped by task

Length	T1	%	T2	%	Total	%
1	9	15.2%	22	23.7%	31	20.4%
2	29	49.2%	47	50.5%	76	50.0%
3	17	28.8%	13	14.0%	30	19.7%
4	3	5.1%	7	7.5%	10	6.6%
5	0	0.0%	3	3.2%	3	2.0%
6	1	1.7%	1	1.1%	2	1.3%
Total	59	100.0%	93	100.0%	152	100.0%

4.5 Usability

Based on the result of the questionnaire, each participant was asked to fill out after the completion of both tasks, the usability score is calculated using the System Usability Scale (SUS) formula [3, p.194]. This calculation leads to an average SUS score of 85.41 out of 100 (higher is better, $\sigma = 6.29$).

For further evaluation, users were grouped by their prior programming experience: All users were asked to answer the question if they have experience in at least one programming language on a Likert-type scale from 1 to 5, with 5 showing the user strongly agrees with this statement, having experience with at least one programming language [3, p.191].

Table 4: Distribution of the programming experience of participants

Experience in Programming	Participants
1 (Strongly disagree)	4
2	3
3	0
4	1
5 (Strongly agree)	4
Total	12

Of all participants, 40% declared an experience in programming of 4 or higher on the Likert scale. Those programming-experienced users lead to an average SUS score of 87.5 ($\sigma = 5.86$), while users inexperienced with programming rated the system 83.9 ($\sigma = 6.59$). Based on discussions with the participants, this difference in rating, in our opinion is the result of the programmers acknowledging the simplicity and usefulness of the system more than non-programmers, because of their experience with the difficulties of programming. Programmers especially mentioned the usefulness of such a system for beginners in programming.

5 CONCLUSION

This paper presented an approach to provide support for end-users to explore third-party APIs, by lowering the barriers introduced by a programming language syntax and an API design and wording unknown a-priori. We achieved that by introducing a programming environment based on a semantic parser that allows to create and compose well-formed programming code using natural language commands.

Furthermore, this paper investigated the efficiency of the approach by conducting a usability study of the environment. The evaluation has shown that programmers, as well as non-programmers, can use the system efficiently after a short introduction and perceive high usability.

Future work on this subject may include an investigation on how to better represent the programming code in the user interface, because as we discovered during the evaluation, most confusion of non-programmers resulted from the representation of the code structure itself. In addition to that, a study with more participants can provide additional insight and will allow for more reliable statements concerning usability.

ACKNOWLEDGMENTS

We express our gratitude to the anonymous reviewers for their valuable feedback as well as the members of AI Systems Manchester for many insightful discussions.

REFERENCES

[1] Imran Sarwar Bajwa, Ali Samad, and Shahzad Mumtaz. 2009. Object oriented software modeling using NLP based knowledge extraction. *European Journal of Scientific Research* 35, 01 (2009), 22–33.
[2] Aaron Bangor, Philip Kortum, and James Miller. 2009. Determining What Individual SUS Scores Mean: Adding an Adjective Rating Scale. *J. Usability Studies* 4, 3 (May 2009), 114–123. http://dl.acm.org/citation.cfm?id=2835587.2835589
[3] John Brooke et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry* 189, 194 (1996), 4–7.
[4] Aditya Desai, Sumit Gulwani, Vineet Hingorani, Nidhi Jain, Amey Karkare, Mark Marron, Subhajit Roy, et al. 2016. Program synthesis using natural language. In *Proceedings of the 38th International Conference on Software Engineering*. ACM, 345–356.
[5] James P Diprose, Bruce A MacDonald, and John G Hosking. 2011. Ruru: A spatial and interactive visual programming language for novice robot programming. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on*. IEEE, 25–32.
[6] Manfred Eppe, Sean Trott, and Jerome Feldman. 2016. Exploiting deep semantics and compositionality of natural language for Human-Robot-Interaction. In *Intelligent Robots and Systems (IROS), 2016 IEEE/RSJ International Conference on*. IEEE, 731–738. https://doi.org/10.1109/IROS.2016.7759133
[7] Eclipse Foundation. 2007. Eclipse IDE.
[8] André Freitas. 2015. *Schema-Agnostic Queries for Large-Schema Databases: A Distributional Semantics Approach*. Ph.D. Dissertation. Citeseer.

[9] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. 1987. The vocabulary problem in human-system communication. *Commun. ACM* 30, 11 (1987), 964–971.

[10] Kelvin Guu, Panupong Pasupat, Evan Zheran Liu, and Percy Liang. 2017. From language to programs: Bridging reinforcement learning and maximum marginal likelihood. *arXiv preprint arXiv:1704.07926* (2017).

[11] IDEA IntelliJ. 2011. the most intelligent Java IDE. *JetBrains[online].[cit. 2016-02-23]. Dostupné z: https://www. jetbrains. com/idea/# chooseYourEdition* (2011).

[12] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The State of the Art in End-user Software Engineering. *ACM Comput. Surv.* 43, 3, Article 21 (April 2011), 44 pages. https://doi.org/10.1145/1922649.1922658

[13] Andrew Jensen Ko, Brad A. Myers, and Htet Htet Aung. 2004. Six Learning Barriers in End-User Programming Systems. In *2004 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2004), 26-29 September 2004, Rome, Italy*. 199–206. https://doi.org/10.1109/VLHCC.2004.47

[14] Stanislao Lauria, Guido Bugmann, Theocharis Kyriacou, and Ewan Klein. 2002. Mobile robot programming using natural language. *Robotics and Autonomous Systems* 38, 3 (2002), 171–181. https://doi.org/10.1016/S0921-8890(02)00166-5

[15] V. Levenshtein. 1965. Binary codes capable of correcting spurious insertions and deletions of ones. Problems of Information Transmission. (1965).

[16] Hang Li. 2011. A short introduction to learning to rank. *IEICE TRANSACTIONS on Information and Systems* 94, 10 (2011), 1854–1862.

[17] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. End-user development: An emerging paradigm. In *End user development*. Springer, 1–8.

[18] Greg Little and Robert C Miller. 2009. Keyword programming in Java. *Automated Software Engineering* 16, 1 (2009), 37.

[19] Hugo Liu and Henry Lieberman. 2005. Metafor: visualizing stories as code. In *Proceedings of the 10th international conference on Intelligent user interfaces*. ACM, 305–307.

[20] Val Marcovic. 2015. YouCompleteMe. http://doc.aldebaran.com/1-14/software/choregraphe/index.html.

[21] Microsoft. 2018. Microsoft Visual Programming Language. https://msdn.microsoft.com/en-us/library/bb483088.aspx.

[22] Rada Mihalcea, Hugo Liu, and Henry Lieberman. 2006. NLP (natural language processing) for NLP (natural language programming). In *CICLing*. Springer, 319–330.

[23] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient Estimation of Word Representations in Vector Space. *CoRR* abs/1301.3781 (2013). arXiv:1301.3781 http://arxiv.org/abs/1301.3781

[24] Alon Mishne, Sharon Shoham, and Eran Yahav. 2012. Typestate-based semantic code search over partial programs. In *Acm Sigplan Notices*, Vol. 47. ACM, 997–1016.

[25] Gonzalo Navarro. 2001. A Guided Tour to Approximate String Matching. *ACM Comput. Surv.* 33, 1 (March 2001), 31–88. https://doi.org/10.1145/375360.375365

[26] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of machine learning research* 12, Oct (2011), 2825–2830.

[27] Jeffrey Pennington, Richard Socher, and Christopher Manning. 2014. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*. 1532–1543.

[28] Aldebaran Robotics. 2017. Choreographe. http://doc.aldebaran.com/1-14/software/choregraphe/index.html. Behaviour Creation Tool for the NAO robot.

[29] Xin Rong, Shiyan Yan, Stephen Oney, Mira Dontcheva, and Eytan Adar. 2016. CodeMend: Assisting Interactive Programming with Bimodal Embedding. In *Proceedings of the 29th Annual Symposium on User Interface Software and Technology*. ACM, 247–258.

[30] Juliano Efson Sales, Siegfried Handschuh, and André Freitas. 2017. SemEval-2017 Task 11: End-User Development using Natural Language. In *Proceedings of the 11th International Workshop on Semantic Evaluation, SemEval@ACL 2017, Vancouver, Canada, August 3-4, 2017*. 556–564. https://doi.org/10.18653/v1/S17-2092

[31] Miguel Gomez Simon and Lara Lorna Jimenez. 2015. Analysis of End-User programming platforms. http://www.diva-portal.se/smash/get/diva2:995396/FULLTEXT01.pdf

[32] Yunwen Ye and Gerhard Fischer. 2005. Reuse-conducive development environments. *Automated Software Engineering* 12, 2 (2005), 199–235.