LIA: A Natural Language Programmable Personal Assistant

Igor Labutov Shashank Srivastava Tom Mitchell
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15217, USA
ilabutov@cs.cmu.edu ssrivastava@cmu.edu tom.mitchell@cmu.edu

Abstract

We present LIA, an intelligent personal assistant that can be programmed using natural language. Our system demonstrates multiple competencies towards learning from human-like interactions. These include (i) the ability to be taught reusable conditional procedures, (ii) ability to be taught new knowledge about the world (concepts in an ontology) and (iii) the ability to be taught how to ground that knowledge in a set of sensors and effectors. Building such a system highlights design questions regarding the overall architecture that such an agent should have, as well as questions about parsing and grounding language in situational contexts. We outline key properties of this architecture, and demonstrate a prototype that embodies them in the form of a personal assistant on an Android device.

1 Introduction

Today's conversational assistants such as Alexa have the capacity to act on a small number of pre-programmed natural language commands (e.g., "What is the weather going to be like today?"). However, advances in semantic parsing and broader language technologies present the possibility of designing conversational interfaces that enable users to instruct (i.e., program) their assistants using language, similar to how humans teach new tasks to one another. For example, if a user wants Alexa to have a new functionality such as *"whenever there is an important email I haven't seen within an hour, read it out to me"*, she should be able to instruct it verbally. This instruction may include explaining what constitutes an *"important email"*. This, in turn, may involve a description such as *"important emails are from colleagues"*, which may require further background knowledge defining *"colleagues"*, *"friends"*, etc. When humans teach other humans, such knowledge is often imparted naturally through explanations, e.g., *"my colleagues would have a CMU affiliation"* or *"Tom is a colleague"*. If AI assistants could be taught in a similar fashion, this could effectively make every computer user a programmer.

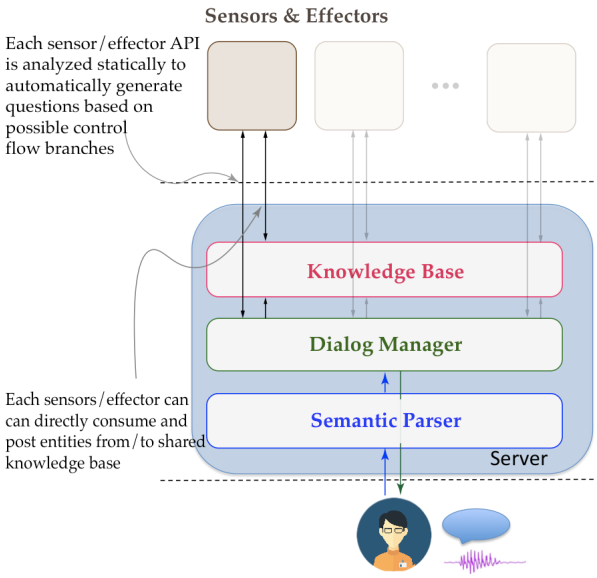

Figure 1: Architecture overview: LIA interacts with the environment through a set of *sensors* and *effectors*, which are mapped to API's of other Android applications. End-users interact with the agent through a text (or voice) interface. User utterances are mapped through a Semantic Parser to logical forms. A Dialog Manager module guides user interactions by grounding logical forms to actions, or asking questions based on possible control flow branches

Towards this end, we present a prototype for a personal assistant, **LIA** (for **Learning from Instruction Agent**), which demonstrates some of these capabilities. LIA resides on a typical mobile Android device. It can perceive the external environment through a set of *sensors* (e.g., sensors for detecting new emails, reading the calendar, reading current time, etc.) and perform actions to change the environment through its *effectors* (e.g., send a message, set an alarm, change the calendar, etc.). The set of sensors and effectors are mapped to functions calls of API's for corresponding Android applications (see Figure 1).

2 Core competencies

LIA demonstrates three core competencies that we consider key for learning from instruction:

2.1 Learning Procedures

Among the main use-cases of being able to teach an agent is being able to define condition-action rules and procedures, such as the following:

```
> If there is an important email then forward to my project team
> Whenever it snows at the night , set my alarm to 30 minutes earlier
> Update my calendar when there is an important meeting request
```

Here, the condition and the effect (action) are highlighted in green and red respectively. The condition in each example requires a check that has to be grounded in the perception sensors. If the condition is satisfied, the required processing consists of calling the execution of actions grounded in the effectors. Giving a conversational assistant the capacity to learn rules verbally opens the possibility of teaching more complex and personalized rules, especially compared to visual programming tools such as IFTTT and Zapier[1]. LIA can ask questions if it cannot parse specifics parts of a user-statement (e.g., if it cannot understand the if-condition, see Figure 2 for an example). Another advantage of a conversational setting is that LIA can take initiative when certain things are left ambiguous by the user (e.g., ask the user what to do if there is a conflict on the calendar for the last rule in the list above) — an issue that cannot be coped with in traditional programming environments.

2.2 Learning World Knowledge

LIA can be taught knowledge about the world by the user (e.g., concepts and ontologies) that can be used as building blocks in teaching new programs.

A key advantage of a conversational interface for teaching new programs is that it allows the user to be naturally expressive about data (i.e., variables/constants) by modelling them after real-world concepts. For example, instead of saying:

```
> If there is an email from Tom, Justine, Oscar or Igor, forward it to Mary
```

LIA allows a user to say:

```
> If there is an email from my project team forward it to my assistant
```

[1] https://ifttt.com/, https://zapier.com/

By relying on the concept of *"my project team"* (instead of listing its members), the second expression is more efficient and natural. It is also better from a programming perspective: if team members change later, the rule will not have to be redefined. LIA enables users to refer to arbitrary concepts such as a *project team* or *colleague*, by declaratively teaching it about them. For example:

```
> Oscar is on my project team
> Everyone on my project team is a colleague
```

The above examples are akin to defining formal data-structures, containing class and field definitions, instance creation and definitions of the natural concept hierarchy (class inheritance). Because the object-oriented programming (OOP) paradigm is designed to model the real world, LIA uses it as the underlying knowledge model that the user can build and modify naturally using language.

2.3 Grounding Knowledge to Perception

Not all knowledge can be easily conveyed through crisp extensional definitions such as in the example of *project team* above. Common concepts such as *important email* or *meeting request* are difficult to declaratively define. In a conventional programming paradigm, the developer may opt to create special functions for grounding such "fuzzy" concepts using machine learning models (i.e., classifiers) that are grounded in attributes of examples (e.g., emails) observed through perceptual sensors.

A conversational programming paradigm offers a natural interface for teaching "fuzzy" and "crisp" concepts alike. Instead of defining hard rules for detecting *important emails* for example, the user may instead opt to provide descriptions that characterize the concept statistically:

```
> An important email will usually be from a colleague's email address
> its subject may contain words like urgent or important
```

By grounding such natural language descriptions to observable attributes of emails, such descriptions can be used to build classification models for concepts such as 'important emails'.

Example Interaction: Figure 2 shows an example interaction exemplifying these abilities in LIA, and also outlining its working. A video demonstration of the system can be seen at http://y2u.be/YfKqpT0apQw. Next, we describe how these abilities are implemented in LIA, and highlight salient features of its architecture.

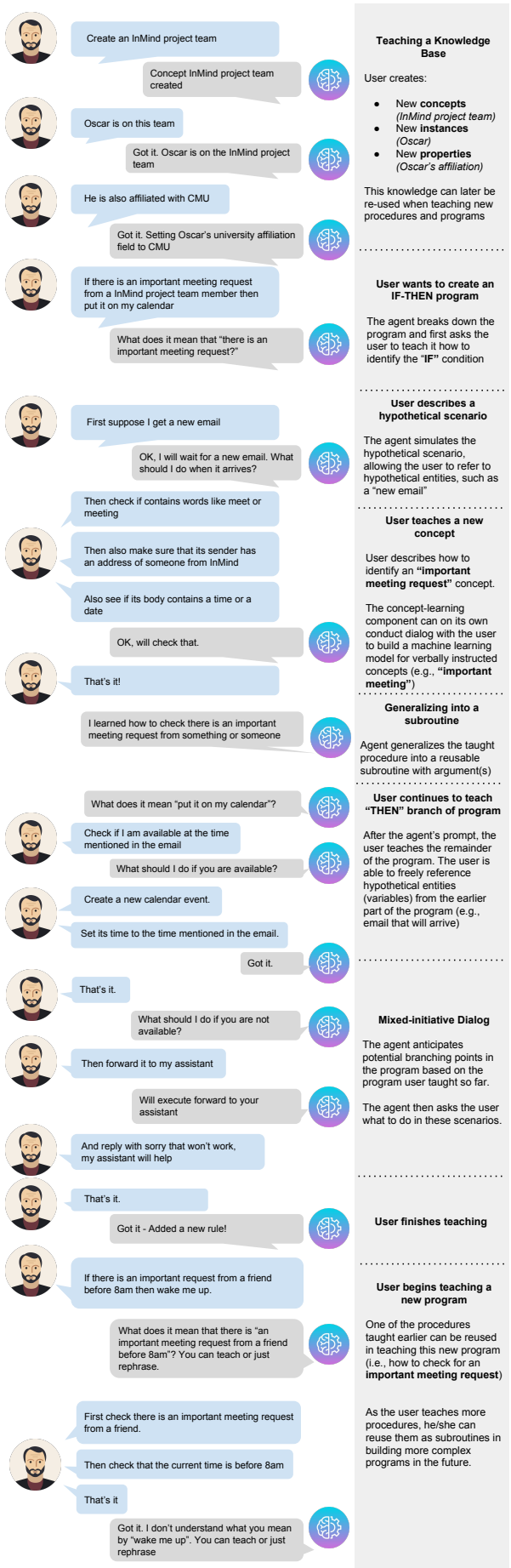

Figure 2: Example interaction sequence with LIA. Annotations on the right summarize different parts of the conversation and outline LIA's working

3 Architecture and System Overview

An instructable conversational assistant can be thought of as a new type of programming interface that allows end-users to compose core functionalities (over domains such as email, calendar, etc.) into programs through natural language dialog. Just like conventional programming languages, this needs answers to design questions such as: "how are new components (functions) imported into the instructable agent", and "how do these components communicate (e.g., what are the data types, how do variables get created and passed between functions)". As examples in this paper illustrate, allowing end-users teach an assistant through conversation brings new challenges to the design of a software architecture that facilitates programming via dialog. We outline five features that we see as fundamental to any system which can be "programmed" through conversation, and describe how LIA implements them:

3.1 Verbally Referencing and Passing Data between Sensors and Effectors

In a conventional programming language, formally declared variables allow one to explicitly store and reference information later in the program. A conversational programming interface needs to allow for a similar mechanism by allowing users to refer and reuse data during instruction through verbal references. For example, consider the following instructions that the user can give while teaching a new procedure *"If there is a meeting request, put it on my calendar"*:

```
> Check if there is a time or date mentioned in
  the message

> Then set the time of the new event to that time
```

This example illustrates the requirement for passing data between two components (*email* and *calendar* APIs), which requires reference resolution on the part of LIA's semantic parser (e.g., which *"event"* did the user refer to in this context?).

LIA solves the problem of interpreting users' utterances and that of resolving references to variables (e.g., *"subject of the received email"* or *"assistant's email address"*) jointly. Reference resolution is context-dependent, and is difficult to solve using rule-based heuristics. The problem of semantic parsing and variable resolution is addressed by LIA using a machine learning

based approach. For example, if the user mentions *"Tom's email"*, and there are multiple contacts named Tom, the agent can use its conversational context (e.g., most recently mentioned entities) and world knowledge to help and resolve the reference – both are naturally incorporated as features; weights for these are learned continuously through interactions with the user.

LIA uses a synchronous CFG-based parser implemented using SEMPRE (Berant et al., 2013), and an underlying frame-based meaning representation (i.e., a frame consists of an intent such as *CREATE_NEW_CONCEPT* and any arguments such as the name of the concept), allowing nested frames for certain intents (currently the *IF_THEN* intent). The parser has an underlying log-linear parameterization of the frame/utterance pairs, with weights that can be learned offline and updated online during interactions with the user. LIA uses the following two classes of features to represent utterance/frame pairs:

- **Lexical/logical form features**: these include indicator features for derivation rules used in the parse, as well as the conjunction of non-terminals in the derivation with the part-of-speech tags spanned by the derivation.
- **Variable resolution features**: these include indicator features that fire if the resolved reference matches only partially to the variable name (e.g., if the user mentions *"affiliation"* rather than *"university affiliation"*), and a feature that indicates whether the variable was mentioned recently in the conversational context.

The *variable resolution* features are very powerful in that they allow the incorporation of external context to help the agent resolve references by relying on the aggregate information from all sensor and effectors of the physical device. For example, a reference to a particular person may be ambiguous when interpreted in isolation, but may naturally resolve to the person who recently sent a text-message or an email. External information such as this, can be incorporated into the *variable resolution* features in a scalable way.

3.2 Generalizing Programs from a Single Example

In conventional programming, explicit functions serve as reusable building blocks and must be expressed via specialized syntax to declare what parts of the procedure can be generalized to different arguments. In a conversational setting, the user is not likely to be explicit about what parts of what they teach should generalize – this knowledge is often implicit based on the context of the taught program. Thus, programs taught via conversation need to be intelligent in automatically generalizing to future invocations with different arguments where appropriate (e.g., if the user taught the agent how to *"tell colleagues to..."*, the same procedure should correctly generalize to *"tell friends to..."*.

When a user teaches a new procedure to LIA, the interaction is always grounded in the specific context within which the user was teaching it. To explain, when teaching how to *"tell my boss that I will be late"*, the user will narrate the sequence of instructions to the agent that repeat arguments from the original command, e.g.,

```
> Set the recipient to my   boss 's email address
> Then set its subject to   I will be late
```

Here, *"boss"* and *"I will be late"* are arguments repeated from the original utterance that is being taught. In a conventional programming language, the programmer would write a function that would explicitly indicate which parts of the procedure are placeholders and would be replaced with arguments in any future invocations of the program. In a conversational setting, the agent must have the capacity to automatically identify what parts of the taught program are placeholders and should be substituted with different arguments in the future. LIA's algorithm is based on Azaria et al. (2016) – it identifies matches between the command being taught and the references to entities made in the program; it then uses this information to store a templated version of the taught program that can be re-used for future invocations with different arguments, e.g.,:*"tell my friends I am on my way"*.

3.3 Define New Knowledge

In a conventional programming language, data structures and their relationships (e.g., inheritance) must be declared formally. On the other hand, LIA infers the data types and relationships declared by the user from natural language statements (e.g., *"most colleagues have a university affiliation"* declares a new field '*university affiliation*', and *"everyone on the cmu team is a colleague"* creates an class-inheritance relation between a member of a cmu team and a colleague). These are identified through a set of manually defined syntactic patterns in the semantic parser.

LIA represents an agent's knowledge in a traditional object-oriented paradigm: the agent's world consists of classes (referred to as *concepts*), and instances (referred to as *objects*). Classes can extend (i.e., be inherited by) at most one other class, while instances can instantiate multiple classes. Further, if a concept (or instance) in LIA extends other concepts, it also inherits all of its fields.

3.4 Grounding New Knowledge in Sensors and Effectors

An important component of an intelligent assistant is the ability to ground language and abstract concepts in observable perception, through sensors and effectors. We envision enabling the agent to learn concepts (such as important emails) from a combination of explanations, and examples of the concept. This is motivated by our recent research on using natural language to define feature functions for learning tasks (Srivastava et al., 2017), and also work on using declarative knowledge in natural language explanations to supervise training of classifiers (Srivastava et al., 2018). Using semantic parsing, we can map natural language statements to predicates in a logical language, which are grounded in sensor-effector capabilities of the personal agent. These may enable the user to:

1. Mention specific attributes that characterize a concept (e.g., define a boolean feature that checks whether an email comes from a colleague)
2. Assert fuzzy statistical constraints specifying relationships between such feature and labels (e.g., 'emails from my colleagues are usually important')

In combination, these capabilities can potentially allow the agent to be taught classifiers for fuzzy concepts from a blend of natural language explanations of these concepts, and labeled or unlabeled data.

Using these explanations and unlabeled data, an automated learner can output a classifier that can predict the class for a new instance. The system can currently be used to train classifiers for a small number of restricted domains. The classifier learning component is currently a standalone module (separate from rest of LIA). We plan to make this publicly accessible in the future.

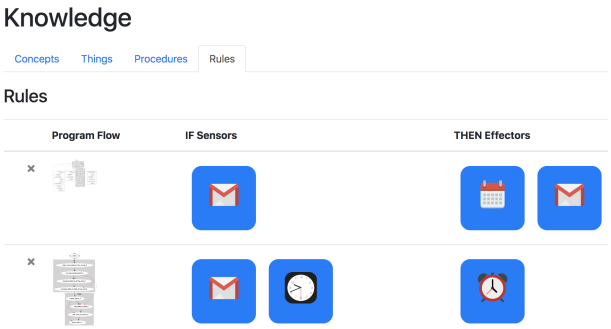

Figure 3: Knowledge View in user interface, which displays procedures taught by a user, along with utilized sensors and effectors

3.5 Mixed-Initiative Interactions

In a conventional programming language, the programmer anticipates all possible outcomes of various API calls made in a program, wrapping these calls with control-flow statements (if/then/else blocks) to account for different return flows. Conversely, end-user programmers must not be required to be explicit about all possible program flows, but rather must inherently be in the form of a mixed-initiative dialog. LIA does this by being pro-active in identifying possible control flow branches based on the instructions the user has provided while teaching. Consider an example:

```
> Check that I am available tomorrow at 2pm
  ...
> What should I do if you are not available?
```

Here, the question creates a control flow branch, from which point on the user instructs a sequence of actions that would be triggered only if the condition that the agent asked about was true. One of the key challenges in providing this mixed-initiative strategy is scaling it to multiple sensors and effectors, where the API for each sensor/effector could trigger a set of potential control flow branches based on the internal execution paths of the individual sensor/effector methods.

LIA's architecture facilitates a generic way of integrating new sensor/effector classes by automatically discovering possible outputs of the API method calls through a static analysis of the API source code. This static code analysis registers this information as possible control flow branches and uses it during the dialog with the user to ask what to do when these control flow branches are reached during execution. Of course, not every possible output of a particular API call (e.g., checking the user's availability) requires asking the user what to