Programmatic Semantics for Natural Language Interfaces

Hugo Liu
MIT Media Laboratory
20 Ames Street 320D, Cambridge, MA USA
hugo@media.mit.edu

Henry Lieberman
MIT Media Laboratory
20 Ames Street 384A, Cambridge, MA USA
lieber@media.mit.edu

ABSTRACT
An important way of making interfaces usable by non-expert users is to enable the use of natural language input, as in natural language query interfaces to databases, or MUDs and MOOs. When the subject matter is about procedures, however, we have discovered that interfaces can take advantage of what we call Programmatic Semantics, procedural relations that can be inferred from the linguistic structure. Roughly, nouns can be interpreted as data structures; verbs are functions; adjectives are properties. Some linguistic forms imply conditionals, loops, and recursive structures.

We illustrate the principles of Programmatic Semantics with a description of Metafor, a "brainstorming" editor for programs, analogous to an outlining tool for prose writing. Metafor interactively converts English sentences to partially specified program code, to be used as "scaffolding" for a more detailed program. A user study showed that Metafor is capable of capturing enough Programmatic Semantics to facilitate non-programming users and beginners' conceptualization of programming problems.

Author Keywords
Programmatic semantics, natural language interfaces, storytelling, brainstorming, case tools.

ACM Classification Keywords
H5.2. User Interfaces: interaction styles, natural language; I.2.7. Natural Language Processing: text analysis.

INTRODUCTION
Amongst the interface options to richly complex systems, natural language input is often considered for its accessibility to non-expert users. However, while it is acknowledged that natural languages are themselves highly expressive, there remains the question of operationalizing that expressivity in light of the limitations of machine interpretation of text.

Still, interface designers have been able to find workable approaches which allow them to reap some of the benefits of natural language input. These approaches are of two general philosophies – 1) the invention of a formal programming language for the interface, whose look-and-feel may be inspired by natural language; and 2) the employment of information retrieval techniques to opportunistically recognize some subset of the semantics contained in a natural language query. The formal language approach assures richer expressivity; however, opportunistic recognition over full-blown natural language is still the most accessible approach for non-expert users.

Currently, opportunistic recognition involves a mish-mash of techniques. Some treat textual interpretation as a problem of statistically classifying a query into *a priori* categories; others apply brittle grammars and templates. Shockingly, a lot of them simply break the user's thoughtfully constructed input back down into keywords. Grammars and templates face problems of robustness and coverage, while statistical classification and decomposition into keywords misses much of the expressivity of natural language, especially its rich procedural relations.

In this paper, we hope to deepen opportunistic recognition of full natural language input by describing a *Programmatic Semantics* for natural language. Programmatic Semantics is a mapping between natural linguistic structures and basic programming language structures, by taking the position that programming *is* storytelling. To grossly oversimplify, noun phrases can be interpreted as data structures, verbs as functions, adjectives as properties. Some linguistic forms imply procedural relations, and others imply looping and recursive structures. Programmatic Semantics has a great potential to help natural language interfaces recover more of language's inherent expressivity, without compromising the interpreter's broad coverage.

PRINCIPLES OF PROGRAMMATIC SEMANTICS
Discussion of the main principles of Programmatic Semantics is divided into 1) syntactic features, 2) procedural features, 3) relational and set-theoretic features, and 4) representational equivalence. These represent a refinement and elaboration over earlier notions presented in (Liu & Lieberman, 2004b; Lieberman & Liu, 2004a). The theoretical conclusions enounced here are grounded in part in our analysis of the user study data of Pane, Ratanamahatana & Myers (2001) on non-programmers' solutions to programming problems like Pacman, which they were kind enough to share with us, and also in part in our practical experience with three generations of iterative

CHI 2005, April 2–7, 2005, Portland, Oregon, USA.
ACM 1-59593-002-7/05/0004.

design and reimplementation of the Metafor story-to-code interpreter.

Syntactic Features

Basic syntactic features of natural language such as *parts of speech, subject-verb-object* distinctions, and *verb-argument structure* can be seen as a regularized vehicle for conveying semantics. The way in which natural language tends to reify concepts as objects with properties, or personify concepts as having capability begins to resemble a style of agent-programming.

The natural role of nouns and noun phrases as objects (e.g. "the martini"), adjectives as properties (e.g. "*sweet* drinks"), non-copular verbs corresponding to functions (e.g. "*make* a drink"), and verb arguments as function arguments (e.g. "give *the drink* to *the customer*") is analogous to the organization of object-oriented programming. Natural language also has a system of inheritance (e.g. "a martini is a *drink* …"), as well as conventions for reference which bring to mind dot notation (e.g. "The customer's age" ←→ customer.age).

Procedural Features

Procedural features of natural language include the expression of conditional rules (e.g. if/then), scripts, and iteration or recursion (e.g. for loops), though Pane *et al.* (2001) report that non-programmers tend not to speak of iteration or looping in explicit terms; instead, they rely on implicit set selection procedures.

We have found that three classes of linguistic constructions account for most conditionals in English: 1) subjunctives, 2) "possibles," and 3) "when." Subjunctive constructions are typically a two-clause construction such as, e.g. *"If* the drink is on the menu, *(then)* make it." Variations include the addition of a third "otherwise" clause, functioning as an "else" statement programmatically. Of course, not all languages have subjunctives, such as Chinese. "Possibles" are the simple modification of a declarative sentence with an auxiliary "may" verb or an adverbial "sometimes," "perhaps," "often," etc. -- "the customer *may* order a sweet drink," "*sometimes* he orders a sweet drink." The difficulty of the "possibles" construction is that the pre-conditional "if" clause is not well specified. This type of ambiguity is common in natural language, but programmatic interpreters might make an educated guess, based on available context, just as people do. The third class of conditional constructions in English is "when," but this is a general scoping construction which can be used dually for declaring if/then clauses, and for specifying the body of a function. In the example, "*when* the drink is sweet, order it," the static situation, "drink is sweet" is topicalized, so the utterance is best realized as an if/then. However, in the example "*when* the customer orders it, the bartender makes it," the function "customer.order(drink)" is topicalized, so the sentence can be best realized as a specification of the body of that function.

To step through a procedural list such as a recipe, an ordering can be constructed using ordinals, e.g. "*first* …, *second*…, *third*…, *finally*..." Alternatively, pairwise orderings can be given, e.g., "Do W, *then* do X. *After* X, do Y, *followed by* Z," and the interpreter must infer the total ordering from these partial ordering (if possible).

Relational and Set-theoretic Features

Perhaps one reason for the absence of explicit looping in natural language is that there already exists basic linguistic constructions that imply a class of procedure which reasons about sets using relational descriptions (e.g. "sweet drinks" as a subset of "drinks"); these set-theoretic constructions seem to supplant the need to narrate looping constructions explicitly. For example, consider the following utterance and the procedure it implies (expressed in Python).

```
The bartender makes a random sweet drink from the menu.
```

```
bartender.make(random.choice(filter(lambda drink:
'sweet' in drink.properties, menu.drinks)))
```

Here, "a random sweet drink from the menu" assumes the same linguistic form as a static object, but it is not a static object at all; rather, it is a *dynamic reference,* an implied procedure which promises to output a pointer to a static object – the procedure employs a series of selectional constraints (e.g. "random," "sweet," "from the menu") to sieve through a "database" of objects. This procedure can either be formed by affixing adjectives to the front of the base object "drink" (as is done in the above example), or, the procedure can be attached, as a complementizer phrase (e.g. "the drink *which is sweet and on the menu*"), or some mix of the two styles is also possible.

Further extending English's set-theoretic discriminatory faculties, a base property such as "sweet" can be coerced into its more discriminating comparative (i.e. "sweeter") and superlative (i.e. "sweetest") forms. Finally, set-facilitation determiners are also a tool of discrimination (e.g. "*all* drinks have", "*some* drinks … while *other* drinks"), but, as with "possible" constructions, these are rather prone to ambiguity in interpretation.

Representational Equivalence

The sort of representational dynamism found in natural language is quite unparalleled by any formal programming language. In Metafor, we always begin by assuming the simplest code representation which can accommodate the facts in the story, dynamically *refactoring* to more complex representations as necessary. For example, the representation of "bar" (given in parentheses) is automatically refactored as each additional fact about bar becomes known:

```
a) There is a bar. (atom)
b) The bar contains two customers. (unimorphic list)
c) It also contains a waiter. (unimorphic wrt. persons)
d) It also contains some stools. (polymorphic list)
e) The bar opens and closes. (class / agent)
f) The bar is a kind of store. (inheritance class)
g) Some bars close at 6pm. (subclass or instantiatable)
```

In formal programming languages, representational revisions b) through g) are potentially quite costly.

Other representational equivalences include morphological equivalences such as *nominalization* – turning any adjective into a noun, e.g. "the drink is *sweet*" vs. "the drink has *sweetness*;" and narrative stance equivalences which allow the same story facts to be inferable through different narrative viewpoints. Consider the following diverse ways to specify that a "bar" object contains "customer" objects:

h) I want to make a bar with a customer. (1st p. programmer)

i) There is a customer in the bar. (3rd p. narrator)
j) I am a customer sitting on a stool. (1st p. customer)
k) The bartender said, "Here is a customer" (mixed person playwright)

The strategy to interpret narrative stance and morphological equivalences is to leverage the already assembled "world model" described by program code to disambiguate utterances, e.g. knowing that there are bar, customer, and stool objects helps us interpret utterances h) through k).

METAFOR: INFERRING CODE FROM STORIES

To illustrate the principles of Programmatic Semantics, we implemented Metafor, a "brainstorming" editor for programs, analogous to an outlining tool for prose writing. As a person types a story into Metafor, the system continuously updates a side-by-side "visualization" of the person's narrative as *scaffolding code* (see Figure 1). This code may not be directly executable, but it illustrates the inherent programmatic structure in English, and is meant to help a person reify her thoughts.

Figure 1. A screenshot of Metafor. Clockwise from the lower left corner, the four windows display 1) the narrative being entered; 2) an interaction log; 3) Metafor's internal representation (not shown to beginning users); and 4) the visualization code, rendered here in the Python programming language.

AN EXAMPLE INTERACTION

Our domain is the world of MOOs (*cf.* Bruckman (1998)), which are popular text-based virtual reality games. MOOs *are* themselves interactive stories, where the characters and even inanimate objects, are programmable. A typical MOO consists of text descriptions of "rooms". Characters in MOOs can be programmed with simple scripts, expressed in an "English-like", though formal, programming language.

Below we present an example of actual Python code outputted by interacting with the Metafor program. This is part of a larger scenario more fully presented in Lieberman & Liu (2005) and by Video Figure 1[1].

When the customer asks the bartender to choose, the bartender makes a random sweet drink from the menu if the customer's age is under 30; or else the bartender makes a sidecar.

```
class customer:
    age = None
       def ask_bartender_to_choose():
        if customer.age < 30:
            bartender.make(random.choice(
                filter(lambda drink:
                  'sweet' in drink.properties,
                  menu.drinks)))
        else:
            bartender.make(sidecar)
```

This example illustrates many of the capabilities we referenced above – creating data structures from noun phrases; functions from verb phrases; recognition of conditionals and implicit iteration over sequences.

USER STUDY

We conducted a very preliminary 13-person user study to test Metafor's impact on brainstorming for a programming task for beginning and intermediate programmers. As would be the case if you were testing an outliner for prose writing, the tool only assists part of the task, so it would be difficult to directly measure performance on the entire task. We opted to let users self-assess the difficulty of the task before and after use of Metafor, on the theory that if it just accomplished increasing users' confidence and perception of efficiency, it would be worthwhile.

Volunteers were MIT undergraduates, 7 intermediate programmers, and 6 beginning programmers. They were asked to implement the basic character behaviors (excluding GUI) of the Pacman game. Initially, we asked them to estimate time for the task (baseline #1). Then they were asked to describe the Pacman program logic on paper in story form, after which they re-estimated the time for the task (baseline #2). Finally, the examiner helped each volunteer to type their hand-written stories into Metafor, sometimes gently rephrased sentences when the story strayed too far from Metafor's linguistic capability (this was not a test of the coverage of Metafor's grammar); after

[1] http://web.media.mit.edu/~hugo/demos/metafor-bartender-simple.mov

doing so, they were once again asked for a time-to-complete-task estimate. The results are shown in Figure 2.

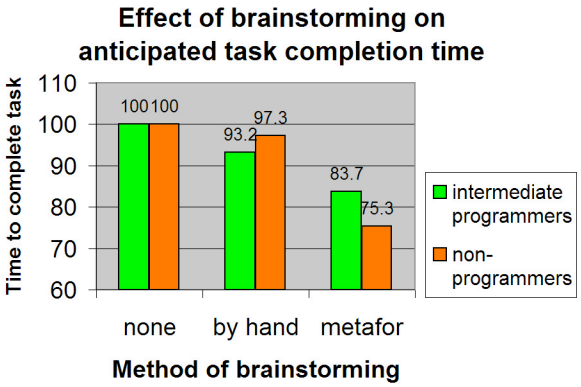

Figure 2: Effect of brainstorming on each volunteer's self-assessment of time-to-complete Pacman task. Times were normalized to 100 to facilitate comparison.

Both non-programmers and intermediate programmers reported that brainstorming with Metafor had a greater positive impact than brainstorming on paper, which in turn had a greater impact than not brainstorming at all. In general, non-programmers felt that brainstorming-by-hand didn't bring them much closer to completing the task (One volunteer said, "I still wouldn't know how to program it."), but they felt that brainstorming with Metafor gave them clearer and more concrete ideas about the programming task. Metafor's advantage over brainstorming-by-hand was less pronounced for intermediate programmers.

On a Likert5 scale (5=very likely, 1=very unlikely), volunteers were also asked how likely they would be to adopt brainstorming-on-paper and brainstorming-with-Metafor in their programming habits. Non-programmers responded that they would adopt Metafor over paper by scores of 4.2 over 3, while intermediate programmers were less enthused with scores of 3.5 over 2.4. The trend though, is that both groups were more enthusiastic about Metafor's interactivity. Three respondents were surprised that their stories translated so directly to programs in Metafor, and one said he would write the story differently knowing now how the computer processes the text. Some respondents remarked that Metafor's interpretation caused them to reflect and think differently about their narrative skills.

RELATED WORK

A companion paper (Liu & Lieberman, 2005) focuses more on the Metafor environment itself, its capability, implementation, and use in programming and education, whereas the emphasis in this paper is on Programmatic Semantics, which we believe has wide applicability in natural language interfaces beyond this particular application.

Related work comes from the fields of automatic programming in AI, and Software Engineering. The best example is Rich and Waters' KBEmacs (1990), which lets programmers express high-level statements of modifications to programming language code, and the editor performs concrete editing of the code automatically. Software engineering methodologies such as the OOD/OOA method of Grady Booch and others have long exhorted programmers to use noun/verb relations in English descriptions as a guide to modeling data/function relations in code, but they have not proposed any way for the machine to help automate such activity.

Two systems from the literature stand out to us as having most directly addressed the problem of translating natural language to code. Tam, Maulsby, and Puerta developed a system called U-Tel (1998) which elicits a story about a task from a person, and allows the person to manually highlight and annotate words in the text with their possible roles. U-Tel also does not produce code directly, but input to a model-based UIMS where the interface can be further specified. Hars & Marchewka's natural language case tool (1996) maps expert-system rules, stated in English, into a yes/no decision flowchart whose nodes are large unparsed natural language utterances.

REFERENCES

1. A. Bruckman: 1998, Community Support for Constructionist Learning. *Computer Supported Cooperative Work, 7*:47-86.
2. A. Hars, J.T. Marchewka: 1996, Eliciting and mapping business rules to IS design: Introducing a natural language CASE tool. In: *Ebert, R.J; Franz, L.: Proceedings, Decision Sciences Institute, 2*, pp. 533-535.
3. H. Lieberman, H. & H. Liu: 2004a, Feasibility Studies for Programming in Natural Language. *Lieberman, Paterno & Wulf (Eds.) End-User Development.* Kluwer.
4. H. Liu & H. Lieberman: 2004b, Toward a Programmatic Semantics of Natural Language. *Proceedings of VL/HCC'04,* pp. 281-282. IEEE Computer Press.
5. H. Liu & H. Lieberman: 2005, Metafor: Visualizing Stories as Code. *Proceedings of IUI'05,* pp. 305-307, ACM Press.
6. J.F. Pane, C.A. Ratanamahatana, B.A. Myers: 2001, Studying the Language and Structure in Non-Programmers' Solutions to Programming Problems. *International Journal of Human-Computer Studies, 54(2)*, 237-264.
7. C. Rich, R.C. Waters: 1990, *The programmer's apprentice*. ACM Press Frontier Series.
8. R.C. Tam, D. Maulsby, and A.R. Puerta: 1998, U-TEL: A Tool for Eliciting User Task Models from Domain Experts. *Proceedings of IUI'98*, pp. 77-80. ACM Press.