Learning Meaning in Genesis

by

Harold Blake Cooper

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

Massachusetts Institute of Technology

June 2009

Author .....
Department of Electrical Engineering and Computer Science
May 22, 2009.

Certified by .....
Patrick H. Winston
Ford Professor of Artificial Intelligence and Computer Science
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Professor of Electrical Engineering
Chairman, Department Committee on Graduate Theses

Learning Meaning in Genesis

by

Harold Blake Cooper

Submitted to the Department of Electrical Engineering and Computer Science
on May 22, 2009, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

GENESIS is an existing software system for understanding and reasoning about language and vision. One of the modules in GENESIS uses about 1000 lines of Java code representing 31 rules to turn English sentences into a variety of more meaningful semantic representations. I reproduced the functionality of these rules by training the existing rule-learning program UNDERSTAND with 43 human-readable examples of English sentences and corresponding semantic representations, resulting in 18 rules. These new rules and the framework for training and using them provides GENESIS with a more robust and extensible semantic parser. This research also led me to make several improvements to UNDERSTAND, making it both more powerful and easier to train.

Thesis Supervisor: Patrick H. Winston
Title: Ford Professor of Artificial Intelligence and Computer Science

Acknowledgements

Mike Klein, for numerous thoughtful conversations, and for creating UNDERSTAND, the impressive system which this paper refines.

Patrick Winston, for teaching the best class I've ever taken, and for creating GENESIS, the *other* impressive system which this paper refines.

Mark Finlayson, for bringing me into the Genesis Group, and for years of guidance and prodding.

Contents

List of Figures

1 Context

1.1 Vision

`Imagine a bird flying to the top of a tree.` Perhaps you can visualize its path through the air, perhaps you're imagining a particular color of bird, perhaps you envision a leafy, deciduous tree, and you can even imagine the roughness of its bark.

Just reading about an event can lead our minds down many of the same avenues as actually experiencing it. And if we actually experience something we can describe it in words to someone else, and they in turn can experience it themselves. We can touch a mug with our eyes closed and imagine what it looks like. We can hear a suitcase dragged across a carpet and imagine what the carpet feels like.

All of these abilities imply that our senses, including vision, speech, touch, and language, are able to share information. When we try to think of a universal language by which these senses could converse, we need to consider what fundamental information such a language transmits, what *semantic representations* it might be made of. We then need to consider how these representations can be *parsed* from sensory information, how they can be reasoned about, and how they can be *transformed* into other representations or back into sensory information.

My thesis research was centered on connecting human language with some of the representations that could form part of a 'universal' semantic language. Specifically, I combined the existing software system GENESIS and the existing program UNDERSTAND such that mappings from language to semantics can be learned by example in GENESIS. Furthermore, I used this new architecture to create a semantic parser which can construct 11 different types of representation from English text, trained with 43 human-readable examples.

1.2 Steps

The Genesis Group at MIT has been studying and using semantic representations for years, and this thesis is further progress in a long series of projects exploring the possibilities of interfacing language, vision, and concise, meaningful representations.

In particular, in this thesis I improved a software system called GENESIS. When you tell GENESIS to `imagine a bird flying to the top of a tree` it creates semantic representations about the spatial path of the bird to the tree, and the decreasing distance and appearance of contact between them. This information is encoded in two of our basic representation types, `trajectories` and `transitions`.

1.2.1 Choosing Representations

In GENESIS, as in its predecessor BRIDGE[1], we are trying to accumulate a variety of cognitively-plausible representations for representing the many semantic facets of reality. Indeed, at the core of GENESIS is the GAUNTLET program, named for how a sensory input is processed by running it through the 'gauntlet' of different representation experts, each one trying to extract its own specialized semantics from the input data. These representations themselves can also be picked up by different experts, enabling translation from one representation to another. See Figure 1 for a screenshot of GENESIS' interface.

Choosing which representations to use is thus the first step toward our vision, because most of GENESIS deals with particular representations. Fortunately, it has been our experience that even a few representations are enough to represent at least *some* of the meaning from most English sentences, so GENESIS is already functional, even as we continue to add new representation types to it. For a list of the representations used in this paper, see Appendix A.

1.2.2 GENESIS' Hard-Coded Semantic Parser

For every semantic representation we add to GENESIS, we want to be able to extract the representation from relevant English sentences. For example, we want to be able to turn sentences about motion into `trajectories` and sentences about change into `transitions`.

To do so, GENESIS first runs the sentence through a *syntactic parser*, such as the Stanford Parser[3], to go from the linear English sentence to a somewhat more consistently-structured form. It then runs the syntactic parse through a *semantic parser* which attempts to extract any of the semantic representations it can.

Currently, this semantic parser is implemented as a set of rules written in Java which transform different sentence types into our different semantic representations in various ways. This works surprisingly well given

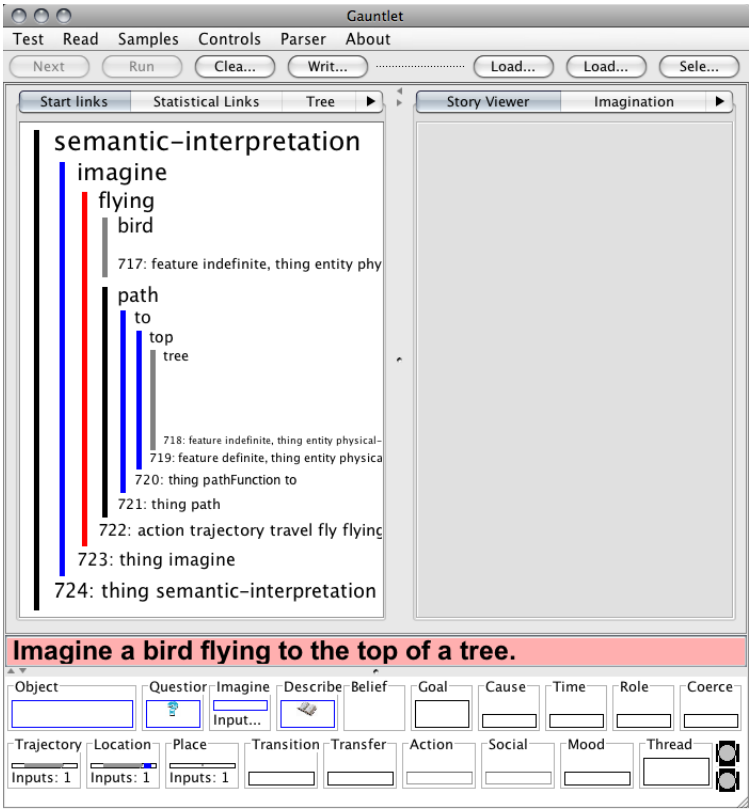


Figure 1: Genesis' user interface. Given the input sentence, the semantic parser has produced the structure shown above the sentence. The Imagine, Trajectory, Location, and Place experts have extracted semantic representations from this structure, as indicated by markings in their widgets at the bottom of the window.

that the rules are applied repeatedly and thus can have unexpected interactions.

The good thing about this approach is that it is completely general. The rules can be arbitrarily complex and sophisticated, and thus any semantic parser is realizable in this way.

The downsides of this approach are that adding a new representation can be quite difficult–requiring significant time and expertise–and that the rules, being arbitrarily complex, can be hard to reason about.

1.2.3 LANCE, a Learning Semantic Parser

The desire for a semantic parser which is easy to extend and easy to reason about led to the idea of a semantic parser which would learn about new representations by example, as opposed to by writing new rules in Java.

LANCE [5] is a semantic parser which learns rules from examples of English sentences and semantic structures corresponding to those sentences. It uses Arch-Learning[8] to update rules based on new examples, generalizing the rules to fit all the known examples.

LANCE was able to reproduce much of the GENESIS functionality using rules learned from 95 examples, but it has never been fully integrated into GENESIS, and generally requires too many examples to be easily trained.

1.2.4 UNDERSTAND, an Improved Learning Semantic Parser

These shortcomings of LANCE inspired the creation of UNDERSTAND [4], another learning semantic parser.

Like LANCE, UNDERSTAND learns from examples of English sentences paired with corresponding semantic structures. It uses several mechanisms to update rules based on new examples, the central technique being known as Lattice-Learning. These mechanisms also allow for the use of very specific negative examples, by correcting the particular piece of bad reasoning in an undesired output.

Figure 2 shows UNDERSTAND making a general rule from a positive example. Figure 3 shows this rule being refined by a negative example.

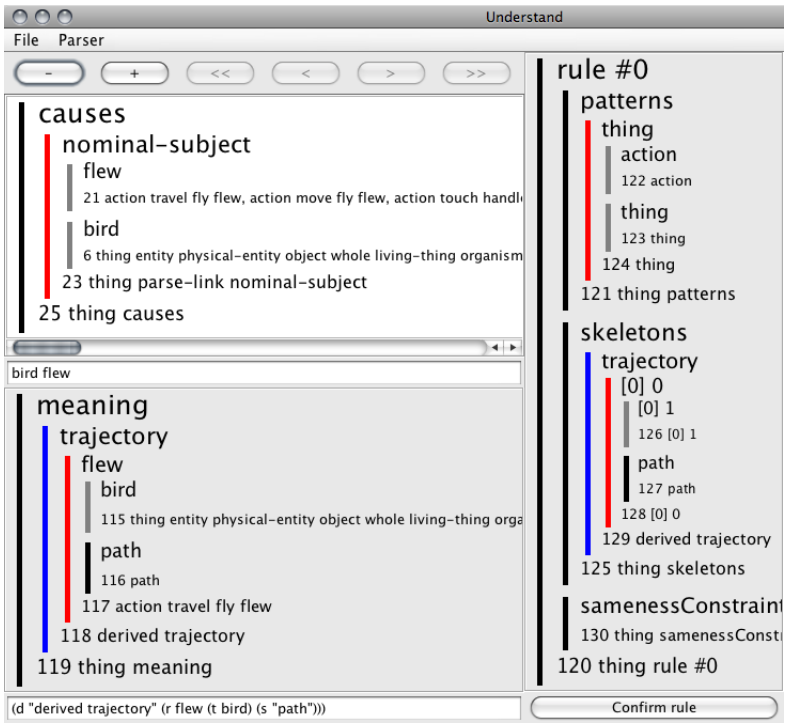


Figure 2: A positive example in UNDERSTAND. The syntactic parse of the input sentence `bird flew` is shown above the sentence. The user has also entered an s-expression describing the desired semantic structure, which is shown graphically above the s-expression. The rule resulting from this example is shown to the right, where `patterns` determines which structures the rule will match and `skeletons` determines how the rule will transform the matched structures.

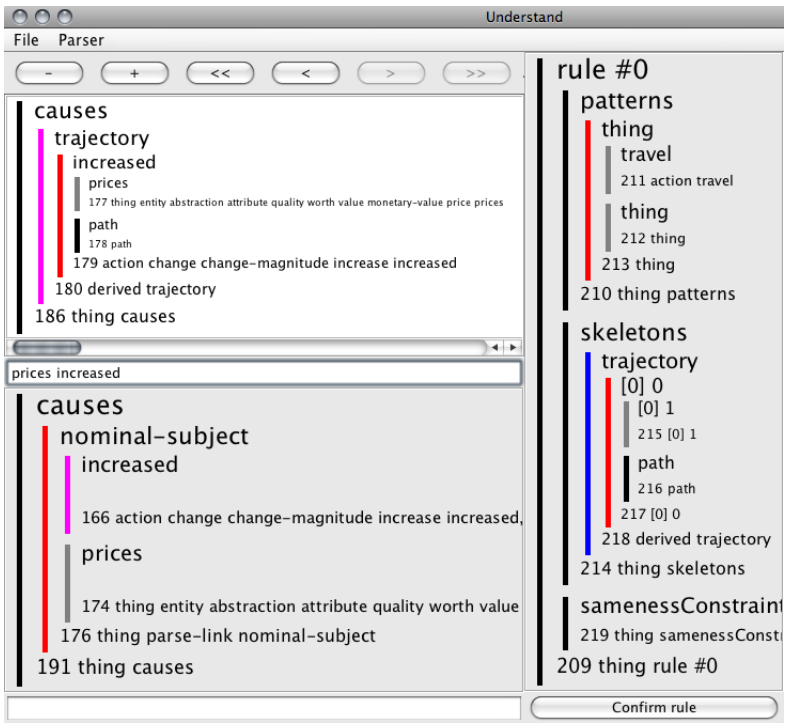


Figure 3: A negative example in UNDERSTAND. If after the positive example in Figure 2 we parse the sentence `prices increased` we see that the new rule matched, producing a `trajectory` structure. We only want `trajectories` to represent motion, however, so we click on the `trajectory` structure to indicate that it is undesired, which turns its bar to pink. The frame below the sentence then shows which structure caused the undesired transformation, and we select the part of the structure which makes the transformation undesirable–in this case `increased`, because it is not a motion verb. Finally, the *updated* rule is shown to the right; its `patterns` now require an action with the type `travel`, so we've successfully restricted `trajectories` to only represent motion.

These very specific negative examples allow UNDERSTAND to generalize very generously based on positive examples, and thus it requires far fewer examples than LANCE to learn comparable rules. For example, UNDERSTAND needed only 12 examples to learn rules similar to the rules learned using 95 examples in LANCE.

1.2.5 This Thesis: Incorporating UNDERSTAND into GENESIS

UNDERSTAND was designed, at least in part, as a learning semantic parser for GENESIS, but it was not initially integrated into GENESIS. In this thesis I integrated UNDERSTAND into GENESIS, and completely reproduced the functionality of GENESIS' hard-coded rules using 43 training examples, as discussed in Section 2.

This was the largest application of UNDERSTAND to date, and revealed some shortcomings of the original program. Thus in this thesis I also improved UNDERSTAND in several ways, and suggests several other ways it could be improved, as discussed in Section 3.

1.2.6 Next Steps: Bootstrapping, Analysis, and a New UNDERSTAND

As the representations and semantic parser in GENESIS are made progressively more expressive and robust, many future paths begin to open.

For the semantic parser, a very interesting next step will be for it to bootstrap on its current knowledge of English to learn to comprehend even more English, all *in English*. For example, if it has rules which associate semantics to the sentence `the man fell to the ground` and it encounters the fact (e.g. in a book or provided by a user) `the man collapsed means the man fell to the ground`, it can then learn a new semantic rule, despite never being explicitly given the desired semantic structure. In this way, once the semantic parser becomes sufficiently robust it will be able to learn more English on its own.

As GENESIS matures it also becomes a powerful tool for examining its own representations. Future work could use GENESIS to explore the occurrence of our chosen representations within large corpora. This might reveal general traits of each representation, as well as connections between them, based on their co-occurrences. These connections, in turn, could lead to new rules for translating between associated representations.

Lastly, as discussed in Section 3.5, the success and power of UNDERSTAND suggests that with several significant changes and improvements it could become an even more impressive system. This includes small gains like smarter learning and easier training, but also fundamental changes like learning to translate both from and *to* linguistic syntax, allowing semantic concepts to be re-described, potentially in a different language.

1.3 News

Using 21 positive examples and 22 negative examples, I've replaced the functionality of the approximately 1000 lines of Java code implementing GENESIS' previous semantic rules. This process is detailed in Section 2 and Appendix A.

To accomplish this I had to improve UNDERSTAND's interface, rule engine, and heuristics. These changes are discussed in Section 3.

An overview of how UNDERSTAND can now be used within GENESIS is given in Appendix B.

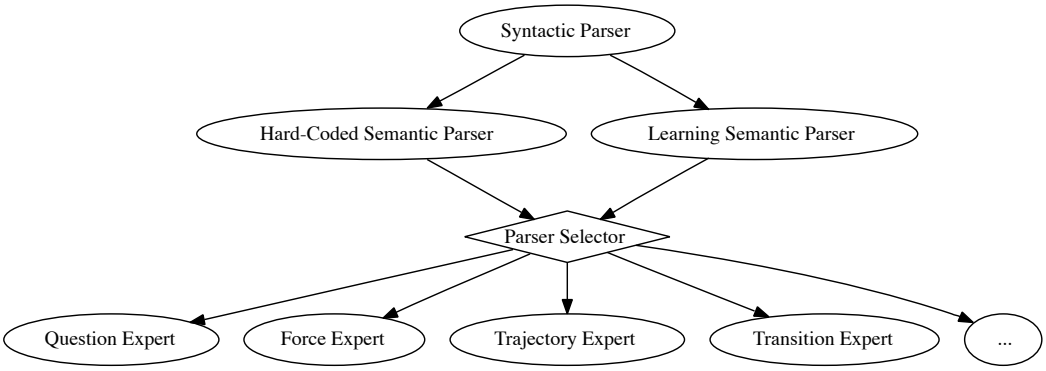


Figure 4: The new dataflow in GENESIS.

2 UNDERSTAND in GENESIS

This section reviews the methods used to reproduce the functionality of GENESIS' hard-coded semantic parsing rules using training examples in UNDERSTAND. Completing this training also required modifications to UNDERSTAND itself, as described in Section 3. The specific examples used and a complete list of the semantic representations involved are given in Appendix A.

2.1 Integration with GENESIS

The hard-coded GENESIS rules were already part of a larger, functioning system, so I wanted to replace them with the learned UNDERSTAND rules in a way that would be unnoticeable to the rest of the system. This required that the logical interface with the new semantic parser be the same as the interface with the old semantic parser, and that the new rules produce the same output format as the old rules, i.e. the same semantic structures.

Replacing the old parser with the new parser was relatively easy. UNDERSTAND's rule engine behaves just like the hard-coded rules, taking an arbitrary structure as input and producing an output structure, so I simply added a switch in GENESIS' user interface allowing the new parser to be chosen. The new dataflow is shown in Figure 4.

The first step towards reproducing the old rules was deciding what exactly their behavior was. Thankfully this was easy, because their author Patrick Winston had provided ample documentation as well as representative test inputs. It's worth noting that if this hadn't been the case, it would have been quite difficult to determine the behavior of the rules based solely on their Java implementation. In UNDERSTAND, on the other hand, the set of positive training examples *already is* an easily-comprehended description of the behavior of the resulting parser.

2.2 A Simple Training Discipline

UNDERSTAND is quite intuitive to use, but while training it on many examples I encountered some subtle pitfalls, and developed some simple principles to avoid them.

2.2.1 Use Minimal Examples

The hard-coded GENESIS rules include a rule called `AbsorbDeterminer`, which essentially removes any determiners like `the` and `a` from the input syntax, to make things simpler for the other rules. Thus `a bird flew to the tree` ends up looking like `bird flew to tree` to all the other rules.

This approach, however, doesn't work in UNDERSTAND, where there is no way to simply get rid of part of the syntax, unless it is part of a larger transformation resulting in some new structure.

Thus, as in the original UNDERSTAND paper[4], one could try simply ignoring any determiners in the input when training. Unfortunately this results in rules which, even though they don't use the input determiners in their output, still expect determiners in their input. So a rule trained on `the bird flew` would also match `a bird flew` but would not match `birds fly`. See Figure 5 for the rule produced by this example.

It may seem reasonable to treat `birds fly` differently because it is a more general sort of phrase than `the bird flew`, but the problem gets worse as the sentences grow. Training on `the bird flew to the tree`, for example, would not only create a rule that requires two determiners, rejecting things like `birds flew to the tree`, but the rule would furthermore require the two determiners to be *identical*, i.e. it would also reject things like `a bird flew to the tree`.

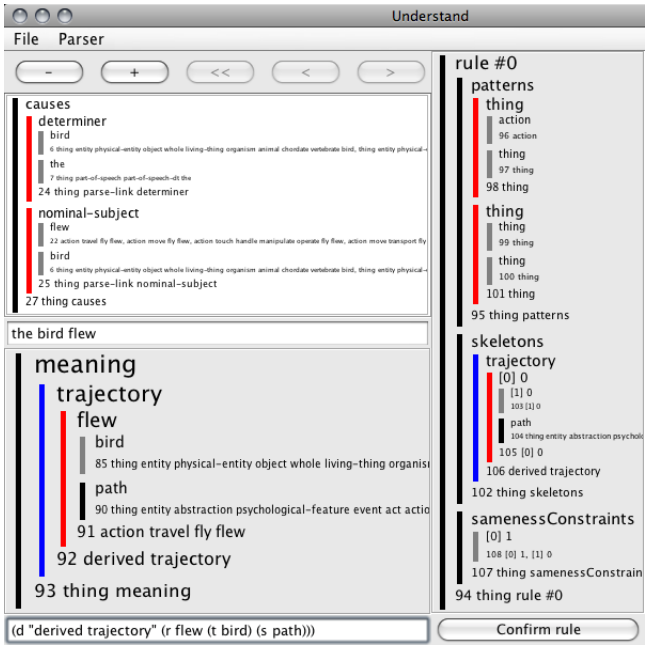


Figure 5: `the bird flew` causes over-fitting. Note that the rule's pattern requires two input structures– which in this case match the parse-links `(bird, the)` and `(flew, bird)`. Thus the rule would not match `birds fly` because its parse has only one parse-link.

To avoid over-fitting the examples in this way, I realized I needed to use minimal examples, which have no extraneous information for UNDERSTAND to falsely consider important.[1] Thus, my training example was `bird flew`, which produces a rule that applies to all of `the bird flew to the tree`, `a bird flew to a tree`, and `birds fly to trees`.

For reasons like this, one should try to use minimal training examples, as part of a general training discipline.

2.2.2 Use Negative Examples First

Another simple but important tenet of training UNDERSTAND in a way that works as expected is that negative examples for a particular sentence should be given before using that sentence in a positive example.

For instance, in Figure 6 we have already taught that `birds flew` should result in a `trajectory`, so at first `prices increased` would also result in a `trajectory`. Thus if we give the positive example associating `prices increased` with a `transition`, the resulting rule will match `trajectories` and change them into

[1] Certain changes to UNDERSTAND itself could also alleviate this over-fitting. See Section 3.5.5.

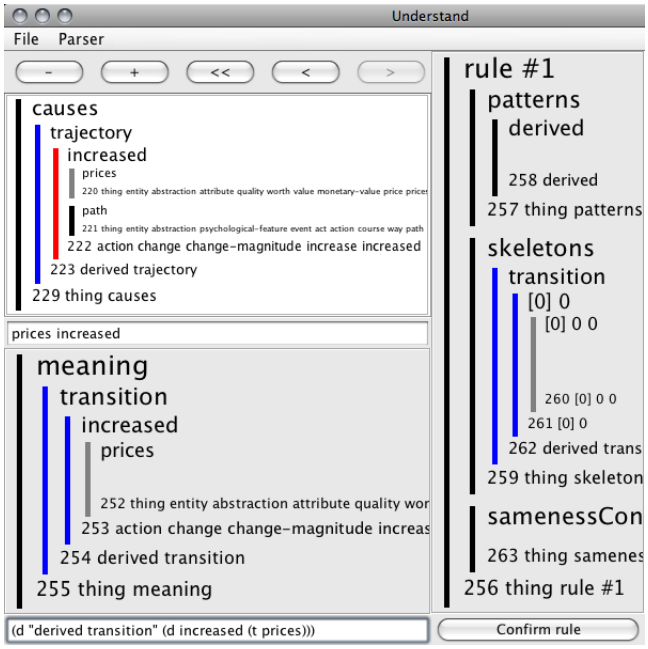


Figure 6: Using a positive example first produces the wrong rule. The system has already been given a positive example associating `birds flew` with a `trajectory`, as in Figure 2. Thus it associates `prices increased` with a `trajectory` as well, so the positive example shown here associating `prices increased` with a `transition` results in a rule that converts `trajectories` to `transitions`, which is not what we want.

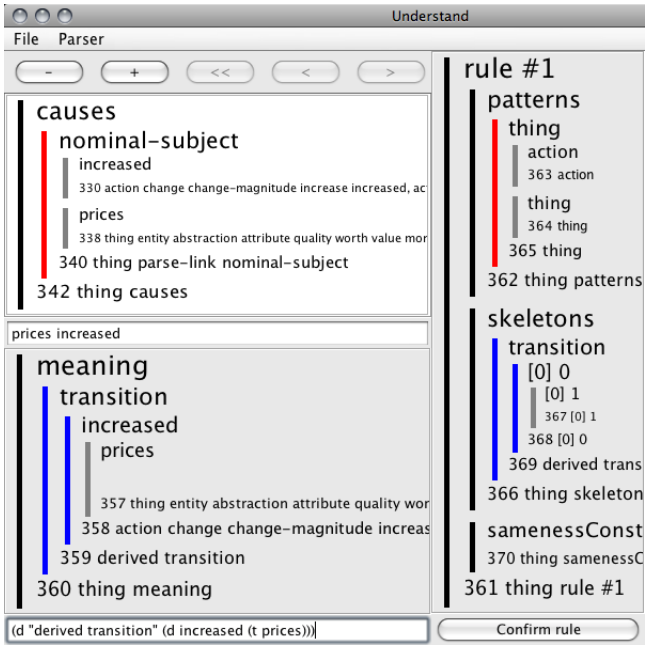


Figure 7: Using a negative example first produces the right rule. The system has already been given a positive example associating `birds flew` with a `trajectory`, as in Figure 2. Thus it associated `prices increased` with a `trajectory` until a negative example broke this association, as in Figure 3. Lastly, the positive example shown here associates `prices increased` with a `transition`, resulting in the desired rule for converting certain syntax to `transitions`. (Note that this resulting rule is actually too general, and would itself need to be constrained by later negative examples.)

`transitions`, which in general is not what we want.

In Figure 7, on the other hand, we have first provided a negative example demonstrating that `prices increased` is not a `trajectory`, and only *then* do we provide a positive example associating `prices increased` with a `transition`. The result is the desired rule mapping syntax to `transitions`.

2.2.3 Retry Positive Examples

A last pitfall to be avoided is that even immediately after teaching UNDERSTAND what a particular sentence should be transformed into, the rule engine won't necessarily transform that same sentence into the desired structure.

For example, in Figure 8 UNDERSTAND learns a very general rule, which is able to match the example sentence in more than one way. Thus when we try running the very example sentence we just taught it with,

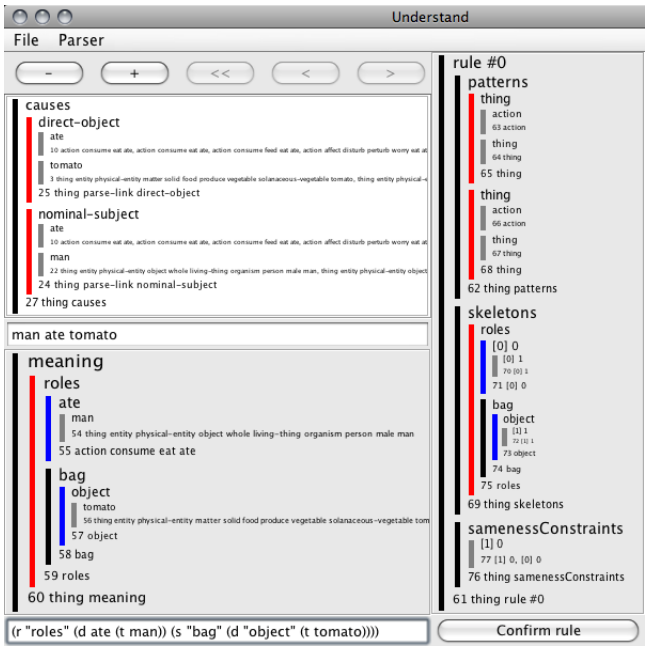


Figure 8: UNDERSTAND doesn't always reproduce its training examples. This positive example for converting `man ate tomato` to a `role` representation initially creates such a general rule that no distinction is made between `man` and `tomato` in the rule's pattern. Thus if we parse `man ate tomato` again we may get the reverse of what we taught it, i.e. the `role` for `tomato ate man`. This is easily remedied with a negative example, which constrains the rule to distinguish between subject and object.

we may get back a different output from that which we taught it. At this point the rule can be strengthened with a simple negative example. We should then try again to ensure that the updated rule now produces the correct output.

Immediately refining each rule in this way avoids more complicated training later on, and thus is worth the small effort. (It might even be a worthwhile addition to the UNDERSTAND training UI for it to automatically re-parse the example sentence every time a rule is learned.)

Consistently following this discipline of only using minimal examples, using negative examples first, and retrying positive examples made training UNDERSTAND very effective and intuitive.

2.3 Learning Higher-Level Representations

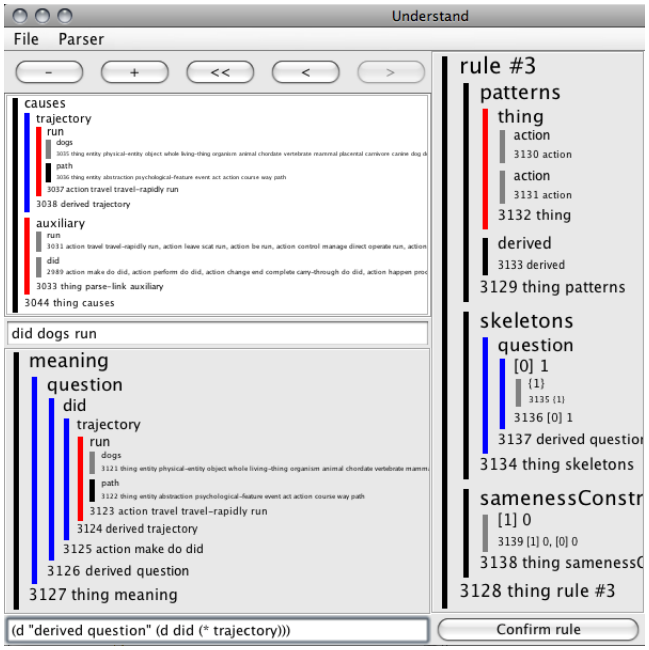
Using the aforementioned training discipline, most of GENESIS' hard-coded rules were easy to replicate by example. One exception, however, was the `force` representation. It provides a good example of using UNDERSTAND to learn *higher-level representations*, i.e. representations that contain other representations.

The `force` representation describes an agent causing something to happen. This 'something' will itself be a representation. For example `the man pushed the couch to the wall` involves the man forcing motion, i.e. it can be encoded as a `force` containing a `trajectory`.

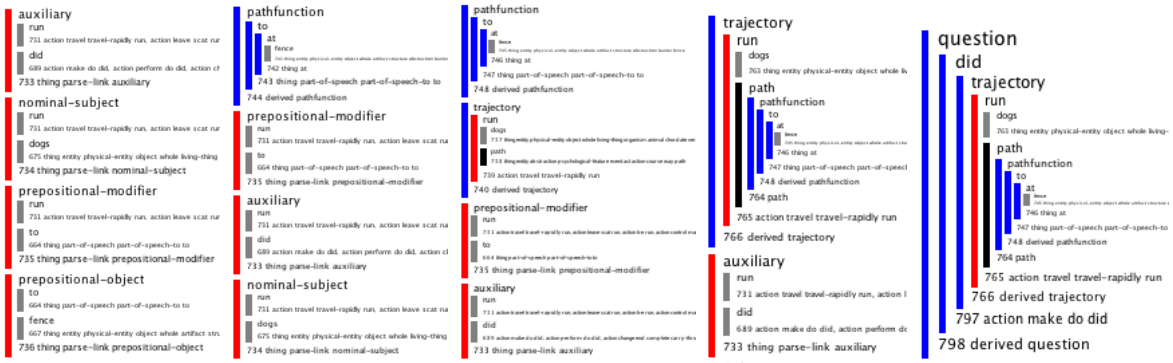
Teaching UNDERSTAND to deal properly with such higher-level representations can be tricky. We want the inner representation to be able to undergo any transformations it might normally undergo, but now it should stay within its parent higher-level representation.

In many cases you can use UNDERSTAND to develop higher-level representations in a very straightforward way. Figure 9 shows how teaching the `question` higher-level representation with one positive example for `did dogs run?` works as expected. Note that in choosing our training example we have used the discipline of *using minimal examples* which allows the learned rule to apply to more complex sentences such as `did the dog run to the fence?`.

If we try the same straightforward training approach with the `force` representation, we aren't as lucky. If we try to use a single positive example for `man forced horses to run`, we notice that the new rule does



(a) The system has already been taught about `trajectories`, so it extracts the `trajectory` from `did dogs run?`. The positive example then teaches it to wrap the `trajectory` in a `question`.



(b) The syntactic parse. (c) A `pathFunction` rule is applied. (d) A `trajectory` rule is applied. (e) A `trajectory` rule is applied. (f) The rule learned in (a) is applied.

Figure 9: A single positive example teaches the `question` higher-level representation. The resulting rule in (a) is general enough to wrap more complex `trajectories` in `questions`, as seen by stepping through the parse of `did the dog run to the fence?` in (b) through (f).

not extend to other `trajectories` such as `the man forced the horses to run to the barn`.

So what's the difference between `did dogs run?` and `man forced horse to run`? It turns out that the syntactic parse for `did dogs run?` includes the same syntactic structures as the syntactic parse for `dogs run`, plus some additional structure representing `did`. Thus the system already knows how to pull a `trajectory` out of `did dogs run?` even if it doesn't yet know what to do with the `did`.

The syntactic parse for `man forced horses to run`, on the other hand, does *not* contain the same syntactic structures as `horses run`, so the `trajectory` rules trained on things like `horses run` don't find the trajectory in `man forced horses to run`.

This is why the straightforward single positive example for `man forced horses to run` fails to create a satisfactory rule: it results in a rule for transforming 'raw' syntax into a `trajectory` wrapped in a `force`, as opposed to creating a rule for wrapping an existing `trajectory` in a `force`.

Once this problem has been clarified, the solution is simple. We use *two* positive examples, the first of which teaches how to get a `trajectory` out the raw syntax of something like `man forced horses to run`, and the second of which teaches how to wrap this `trajectory` in a `force`. This process is shown in Figure 10.

Training UNDERSTAND to deal with higher-level representations demonstrates both its power and subtlety. As with the `question` representation, it can handle most higher-level representations just as easily as anything else, and with some careful thought it is easy to train it on more complex higher-level representations like `force`.

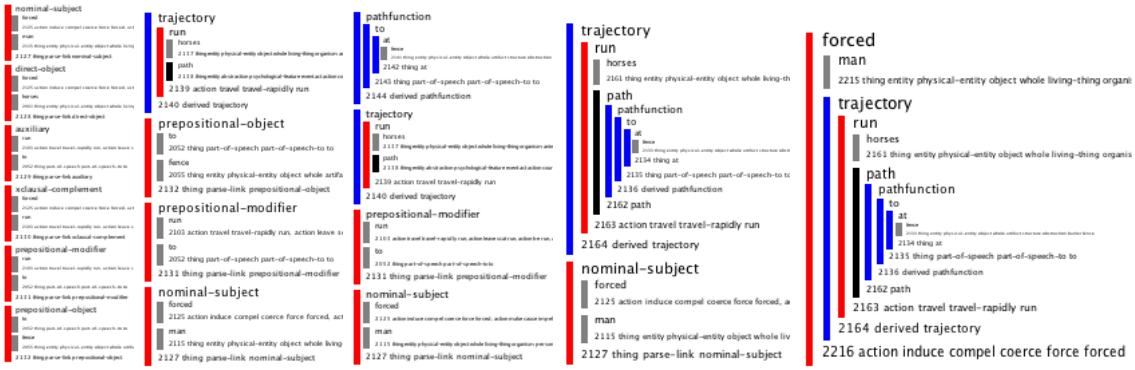
2.4 Hard-Coded Rules Versus Learned Rules

By integrating UNDERSTAND into GENESIS' dataflow and then training it while keeping in mind the discipline discussed above, I was able to re-produce the previous hard-coded rules with rules learned from a small list of human-readable examples.

The hard-coded rules consist of about 1000 lines of Java code, including comments, comprising 31 rules. The strength of the hard-coded approach is practically unlimited generality, because any code could be used. This is also a weakness, in that the rules and their interactions can become arbitrarily complex. The other

(a) The first example teaches UNDERSTAND to extract a root-level `trajectory` from `man forced horses to run`, ignoring the `man forced` syntax.

(b) The second example creates a rule for combining the `trajectory` now found in `man forced horses to run` with the leftover `man forced` syntax to create a `force` representation.



(c) The syntactic parse.

(d) The rule learned in (a) is applied.

(e) A `pathFunction` rule is applied.

(f) A `trajectory` rule is applied.

(g) The rule learned in (b) is applied.

Figure 10: Two positive examples teach the `force` higher-level representation, in (a) and (b). The sentence `the man forced the horses to run to the fence` is parsed in (c) through (g), using these new rules and previously-learned rules for combining paths and `trajectories`.

difficulty is that writing rules can be a difficult process, requiring time and expertise.

The learned rules consist of 21 positive examples and 22 negative examples resulting in 18 rules. The strength of this approach is that the training is a relatively intuitive process, and produces a human-readable training document consisting of examples not dissimilar to examples you would use to teach a *human* about the particular representations. Another strength is the uniformity of the rules, all of which can be graphically represented in the same way. This uniformity of rules is also a weakness, in that any transformation that cannot be expressed as such a rule is simply impossible to perform in UNDERSTAND.

It remains to be seen whether UNDERSTAND's restricted rules will be able to capture every representation we might ever want to add to GENESIS, but the fact that I was able to reproduce all of GENESIS' functionality so far provides reason to be optimistic.

Of course, UNDERSTAND isn't perfect, and even to learn the rules taught in this thesis it had to be improved somewhat, as described in Section 3. Section 3.5 also discusses some further changes to UNDERSTAND that might make it even more powerful.

3 Furthering UNDERSTAND: A Further Understanding

3.1 Step-by-Step Reasoning

3.1.1 Generalizing Negative Examples

UNDERSTAND learns by example, so it is very important to be able to easily provide it with both positive and negative feedback. The means of providing positive feedback are relatively unconstrained, because any pair of an input structure and a 'correct' output structure can be provided as a positive training example. Negative examples, however, must refer to a specific point at which UNDERSTAND's reasoning was incorrect.

Thus for negative examples it is important to be able to access and refer to as much of the reasoning used by the rule engine as possible. For example, suppose we train UNDERSTAND to generate a `transition` describing the decrease in distance between two objects every time it sees a `trajectory` saying that one object moved towards the other. Thus `the bird flew to the tree` would be transformed first into a `trajectory` and then into a `transition` representing `the distance between the bird and the tree decreased`.

Furthermore, suppose that the `trajectory` rules were not yet fully trained, and in particular `the man appealed to the court` will also be transformed into a `trajectory`. Normally this would easily solved by a simple negative example. But now this intermediate failure is hidden because the final output produced by UNDERSTAND is a `transition` representing `the distance between the man and the court decreased`. We can't make a negative example from this `transition` because that would modify our new rule mapping `trajectories` to `transitions`, which is not actually the rule at fault.

Thus we need a way to construct a negative example on the intermediate `trajectory` structure that is no longer visible in the final output. This sort of scenario may sound contrived, but in my experience it happens more and more as the number of rules and examples grows. Being able to step the rule engine back and forth allows us to easily solve the problem.

3.1.2 Implementing Stepping

In the original implementation of UNDERSTAND's training interface, negative examples could only be based on structures in the final output. In creating a full GENESIS rule-set, however, I came upon situations like the example above, where I needed to correct the engine for a mistake that was no longer visible.

Thus one of my additions to the system is the ability to step through each iteration of the rule engine, allowing negative examples to be based on any intermediate structure.

This is implemented by modifying the rule engine to store the *cause* of each universe. This cause consists of the previous universe that the new universe was created from, and, for reference, the rule that created it.

Once these causes were stored, I added standard temporal navigation buttons to the training interface: << rewinds to the initial universe, i.e. the syntactic parse, < steps back one iteration, to the cause of the current universe, > steps forward one iteration, and >> displays the final output universe. This user interface can be seen in Figures 2 and 3.

In addition to enabling more powerful negative examples, step-by-step processing is also immensely useful for analyzing and reasoning about what the rules are actually doing and how they are interacting. For example, it lets us watch the progression of transformations involved in processing complex sentences, which was used to make Figures 9 and 10.

3.2 Finding the Right Heuristic for Finding the Right Result

The rule engine computes all possible applications of rules, resulting in multiple 'universes' of structures. A universe is simply a collection of root-level structures such as parse links, representations, or a combination of both. When it has applied every possible series of rules, however, it chooses a particular 'best' universe to return to the user. This is done using a heuristic which, based on structural properties and potential 'priming' in favor of certain outcomes, compares universes and chooses the maximal one.

The original heuristic used was called DEEPEST-AVERAGE. To compute the score for a given universe, it computes the depth of each structure in the universe and averages them. This emphasizes *deep* structures and *few* structures. In particular, any un-processed structures left from the original English parser are assigned

(a) Universe A. (b) Universe B.

Figure 11: Using the DEEPEST-AVERAGE heuristic, universe B would be chosen over universe A.

a depth of zero, dramatically reducing the average depth of the universe.

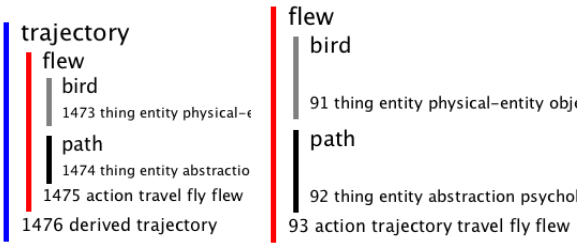
In Figure 11 for example, using the DEEPEST-AVERAGE heuristic, universe B would be chosen over universe A.

While writing the full set of GENESIS rules, I began to realize that, at least for these rules, this heuristic was not ideal. There's no question that 'raw', 'undigested' structures from the English parser should reduce the score of the universe that contains them, but DEEPEST-AVERAGE seems to over-emphasize this consideration, making it quite unlikely that a universe with any undigested structures will ever be chosen as the final structure. As in the example above, however, and in many of the GENESIS rules, this results in the wrong output.

Thus I implemented a new heuristic, which I'll call DEEPEST-HEAVIEST-THINNEST, which emphasizes the desirable traits of GENESIS representations, while still achieving the main goal of DEEPEST-AVERAGE, i.e. to discourage undigested structures.

DEEPEST-HEAVIEST-THINNEST is relatively simple, and almost completely described by its name. It chooses the universe with the greatest depth. This is different from the greatest *average* depth, in that it is not decreased by the shallower structures in the universe, as was the case for DEEPEST-AVERAGE in Figure 11. In Figure 11, DEEPEST-HEAVIEST-THINNEST would choose universe A because it is deeper than universe B.

When there are two universes with the same depth, which is very common with our representations, the *heaviest* is chosen, i.e. the one with the most nodes in its structure. Just as DEEPEST-AVERAGE treated



(a) (b)

Figure 12: The `derived` keyword enables UNDERSTAND to modify threads. Thus structure (a) is equivalent to structure (b).

raw syntactic structures as having zero depth, DEEPEST-HEAVIEST-THINNEST treats them as having zero weight. Thus unparsed syntax is still discouraged, but less strongly.

Lastly, if two universes have both the same depth and the same weight, then the *thinnest* is chosen, i.e. the one with the fewest root structures. This prefers one large representation to several smaller ones.

3.3 Using the `derived` Keyword to Modify Threads

One of my primary goals while integrating UNDERSTAND into GENESIS was to make the switch between the existing hard-coded rules and the new learned rules essentially invisible to the rest of the system. In particular, I wanted the output of my rules to be indistinguishable from the output of the old rules.

GENESIS' hard-coded rules, however, do one thing that was initially completely unsupported by UNDERSTAND, which is that they modify *threads*. Threads[7] are the type vectors used in our representations and on which UNDERSTAND's Lattice-Learning operates to learn about what types a rule can be applied to.

For example, `flew` is assigned the simple thread, taken from WordNet[6], `action travel flew`. When GENESIS' hard-coded rules transform `the bird flew` to a `trajectory` they change the thread of `flew` in the output structure to `action trajectory travel flew` so that `flew` now represents a `trajectory`.

There is no syntax for this sort of change in UNDERSTAND, so I implemented special treatment for the keyword `derived` such that wrapping a structure with the thread `derived <new-type>` around another structure results in `<new-type>` being added to the thread of the inner structure. For example, Figure 12 shows the use of `derived` to represent the `trajectory` corresponding to `the bird flew`.

The implementation of the `derived` keyword actually involved no modifications to UNDERSTAND at all. Instead, there is simply a translator which collapses all of the `derived` structures and performs the thread modification just before the rest of GENESIS sees the output of UNDERSTAND. A better solution would probably involve new syntax for UNDERSTAND, as discussed in 3.5.4.

3.4 Multiple Parsers

Recently, GENESIS incorporated a new English-language parser, the START parser.[2] The output of START is already very semantically-oriented, in sharp contrast to the syntactic output of the Stanford parser.

I modified the UNDERSTAND program to support both parsers. This required no modification to the rule engine itself, because it can deal with any structured input, and thus has no bias towards the Stanford parser over START. The only necessary modifications were interface additions for choosing a parser, and file-format changes for remembering which parsers were used for which training data.

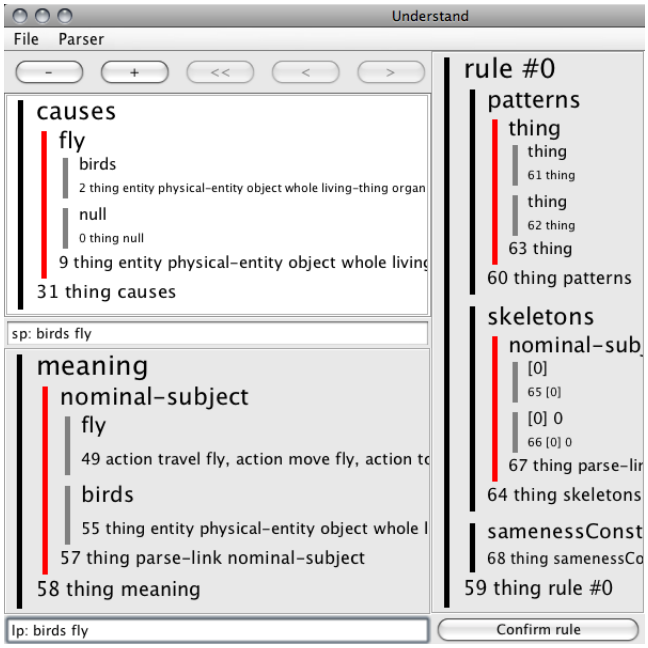
Of course, the learned GENESIS rules expect Stanford parser input structures, so the START parser input isn't matched by any of the old rules. There are two approaches to incorporating START into GENESIS, such that the rules know how to go from START inputs to our semantic representations. The obvious way is to create training examples very similar to those used for learning the rules for the Stanford parser, resulting in two subsets of rules, one for each parser.

The other, perhaps more interesting, approach is to write rules for translating from the START input syntax to the Stanford input syntax. This is somewhat like 'translation' from an unknown foreign language (START syntax) to an understood language (Stanford parser syntax), enabling a system to understand the unknown language, but only via the known language, as opposed to directly.

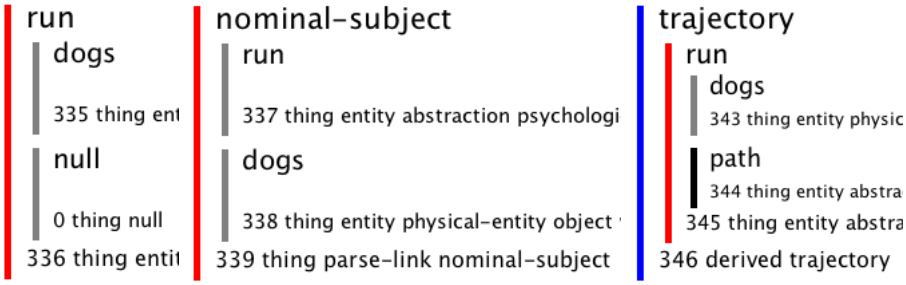
Figure 13 shows an example of how this translation could work in UNDERSTAND.

3.5 Recommendations

Integrating UNDERSTAND into GENESIS confirmed the impressive capabilities of the system, but also revealed several subtle shortcomings. In this thesis I resolved the needs for step-by-step observation of the rule



(a) A simple translation rule from START parser (`sp`) syntax to Stanford link-parser (`lp`) syntax.



(b) First, the raw START syntax.

(c) Next, our new rule from (a) translates the START syntax to Stanford syntax.

(d) Lastly, a `trajectory` is produced by a standard rule that only knows about Stanford syntax.

Figure 13: Translating between different syntactic parsers. We first train UNDERSTAND to translate the START parser syntax for `birds fly` to the Stanford parser syntax, in (a). We then load all of the semantic rules we've already written for the Stanford parser (not shown). Now when we input `dogs run` using START, UNDERSTAND executes steps (b) through (d). Thus we have semantically parsed a trajectory with the START parser, despite having never trained any semantic rules using the START parser.

engine and a better heuristic for choosing final parses. But my experience applying UNDERSTAND to a substantial problem has suggested several further ways in which the program could be improved. Thus a small contribution of this thesis is the following set of informed recommendations for improving UNDERSTAND.

3.5.1 Multiple Outputs

There are many sentences which should result in multiple semantic representations. For example, `the car drove into the tree` might lead UNDERSTAND's rule engine to two different universes, one containing a `trajectory` of the car's motion, the other containing a `transition` describing the decrease in distance between the car and tree, followed by the appearance of contact. In this case it would be ideal for both universes to be chosen for the final output.

Because the heuristic decides what is chosen for final output, choosing multiple outputs would require a different heuristic or different use of heuristics, as discussed below.

3.5.2 A Better Heuristic

The fact that I had to implement a new heuristic for UNDERSTAND to properly implement GENESIS' rule system suggests that the current design is too dependent on its particular heuristic. Thus we should seek either a 'better' heuristic which almost always chooses the right output, or a more permissive heuristic which allows multiple universes to be chosen for output.

It seems unlikely that there is a 'generally better' static way to judge semantic structures. Perhaps the best candidates would be based on information content (e.g. minimum description length), because most representations will probably try to lose as little information as possible while adding structure. Even this assumption, however, depends somewhat on the design of the semantic representations.

So for UNDERSTAND to work with any structures, a *learning* heuristic might be the only way to accomplish the behavior desired by the designer of the representation structures. The training data for a learning heuristic are already provided: it could use the very same examples that the rule engine is already being given. It is already the case that even immediately after providing UNDERSTAND with an input/output pair in the form of a positive example, it can produce a different output given the same input, because the

heuristic ends up preferring a different rule than the one that was just learned. This is a perfect moment for a learning heuristic to update itself, to ensure that the desired output is chosen instead.

If a consistently 'better' heuristic proves evasive, or for other reasons as discussed above, it might be desirable for UNDERSTAND to allow multiple universes as output.

For example, one of the many possible approaches to this problem would be to use 2-means clustering on the heuristic scores. This would group universes into two clusters: those chosen for output (which would always include the highest-scoring universe) and those not shown as output (which under most heuristics would always include the original 'raw' syntactic parse).

3.5.3 ...or No Heuristic

Of course, an even simpler way to allow multiple outputs is to have no heuristic at all, and output every universe generated by the rule engine. This essentially shifts the burden of choosing the desired representation to the user of UNDERSTAND, allowing tailor-made heuristics to be applied to their particular structures.

In GENESIS in particular, this latter option of no heuristic may actually be the best approach. This is because GENESIS already has multiple 'expert' modules looking for particular representations; anything that no expert wants is ignored. Thus no undesired structures would be consumed, but no desired structures would be blocked by UNDERSTAND's built-in heuristic.

3.5.4 Syntax for Modifying Threads

As discussed in Section 3.3, I introduced the `derived` keyword to enable UNDERSTAND rules to modify threads. This approach is an expedient however (read: hack), so a true addition to the UNDERSTAND syntax would be preferable.

This could be accomplished by adding thread functions to the existing s-expression syntax used in the program. For example, adding the type `trajectory` to the thread for `flew` might look like:

```
(d (+ "trajectory" flew) ...)
```

This would be equivalent to what would currently be written as:

```
(d "derived trajectory" (d flew ...))
```

3.5.5 Smarter Patterns and Sameness Constraints

After a rule is created, the only parts of it that are updated by new examples are the threads it matches against. This does an extremely good job at refining precisely which sorts of verbs, nouns, and representations a rule should apply to.

The remaining parts of the rule, however, are never changed after they are first created by a positive example. These remaining parts include the structure of the rule's *patterns*, which determine what input structures match the rule, and the rule's *sameness constraints*, which further filter the possible inputs by ensuring that certain parts of the input structure are the same as other parts of the input structure.

Often these input constraints work very well, but sometimes they over-fit the rule to the initial example that created the rule. For example, as discussed in Section 2.2.1, if the word `the` occurs in more than one place in the input sentence, then a sameness constraint will be established that *requires* two of the same articles to appear. Thus a rule trained on `the bird flew to the tree` would also match `a bird flew to a tree` but not `a bird flew to the tree`. Indeed, these 'greedy' sameness constraints are one of the main reasons for the need to *use minimal examples*, as espoused in Section 2.2.1.

Over-fitting to one example is perfectly reasonable behavior, but adding the ability for further examples to change the patterns and sameness constraints of rules could potentially make UNDERSTAND even more powerful and an even faster learner.

3.5.6 Two-way Transformations

Positive examples in UNDERSTAND are essentially equivalences, saying that, e.g. a particular English sentence is equivalent to a particular semantic structure. Thus it would be reasonable, given the semantic structure, to produce the English sentence.

In particular, every training example could be run once in each direction, producing two rules.

One intriguing application of this might be to machine translation–if you have examples connecting a common

semantics to two different languages, you could then do something like the following: given English syntax, the English rules would transform it into a semantic representation, which the reverse-Chinese rules would then transform into Chinese syntax. Thus with a good enough English parser and a good enough Chinese parser you could create a English-to-Chinese translator.

4 Contributions

In this thesis I made several different contributions, all centered around semantic parsing in the GENESIS system.

4.1 Improved GENESIS

I improved the GENESIS system by replacing its hard-coded semantic parser rules with new rules based on human-redable examples. These new rules are easier to extend and easier to comprehend because they are based on English examples and all fit into a restricted form that can be examined graphically. Specifically, 43 examples were used to generate 18 rules, replacing about 1000 lines of Java code implementing the old rules.

The integration of UNDERSTAND into GENESIS and the issues involved in training the new semantic parser are discussed in Section 2. The specific examples used are given in Appendix A.

4.2 Improved UNDERSTAND

I improved the UNDERSTAND program in four ways. I gave UNDERSTAND the ability to step through each stage of a semantic parse, enabling the creation of more specific examples. I improved the heuristic used to determine the final output of the parser. I gave rules the new ability to modify the *threads*, i.e. the types, of objects being parsed. Lastly, I added support for the START syntactic parser.

These improvements are detailed in Section 3.

4.3 Elucidated Training Methods

I provided a thorough discussion of the difficulties encountered while training GENESIS' new semantic parser, and documents the solutions to these problems. In particular, I proposed a simple discipline for training UNDERSTAND in a way that avoids complex problems as the system learns more and more rules. Another contribution is a guide to using the new software, detailing the workflow for adding new rules to GENESIS.

The discipline used for training GENESIS' new semantic parser is given in Section 2. The process of adding new rules to GENESIS is laid out in Appendix B.

4.4 Pointed to Future Work

I provided several goals for the next version of UNDERSTAND, as well as some vision for the next steps in semantic representation research. Using UNDERSTAND to create a full semantic parser in an existing system allowed me to notice several ways in which UNDERSTAND could be made more powerful and useful. Working with GENESIS led me to speculate on the some of the potential research that could be conducted using it.

The recommended improvements to UNDERSTAND are given in Section 3.5. Some pointers to future work in GENESIS are provided in Section 1.2.6.

A Semantic Representations and UNDERSTAND Examples

This appendix details the specific semantic representations used in this thesis and provides the specific training examples used to build a semantic parser for handling these representations.

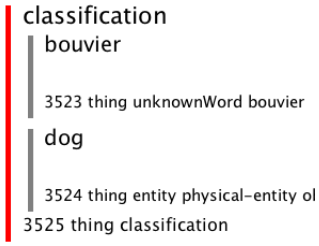
A.1 Semantic Representations Used in GENESIS

This section lists the 11 different semantic representations currently supported by the semantic parser built in this thesis. These representations appear throughout the training examples given in the next section.

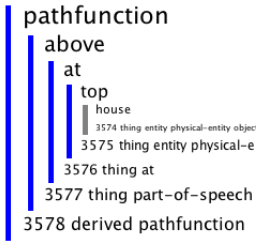
Each representation is briefly described and then an example structure (usually based on a positive example from the next section) is shown. They are given in the order they were taught in.

More semantic representations are already being added to GENESIS. As new representations are added, it is relatively easy to train the semantic parser with a few new examples to enable support for these new representations.

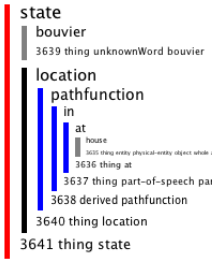
A `classification` simply represents a definition of an unknown word.
For example, `a Bouvier is a dog` corresponds to:



A `pathFunction` represents part of a spatial path.
For example, `above the top of the house` corresponds to:

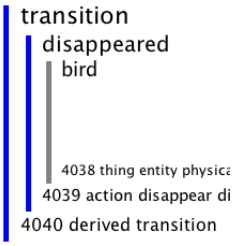


A `state` represents a current fact.
For example, `the Bouvier is in the house` corresponds to:



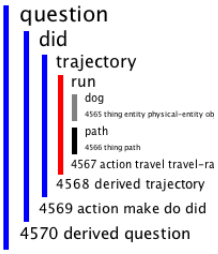
A `trajectory` represents an object moving along a path.
For example, `the boy ran from the house to the tree` corresponds to:

A `transition` represents a change, such as a decrease or increase or appearance or disappearance.
For example, `the bird disappeared` corresponds to:

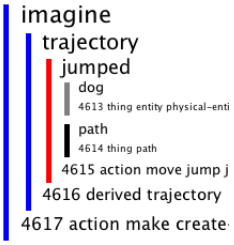


A `cause` represents causality or temporal relation. It is a higher-order representation, in that it can contain other representations.
For example, `the robin disappeared because the kestrel flew over the tree` corresponds to:

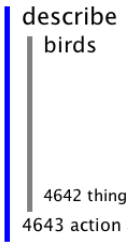
A `question` is a simple higher-order wrapper representation for inquiring about the truth of another representation.
For example, `did the dog run?` corresponds to:



An `imagine` is a simple higher-order wrapper representation for asking Genesis to imagine a particular scenario.
For example, `imagine the dog jumped` corresponds to:

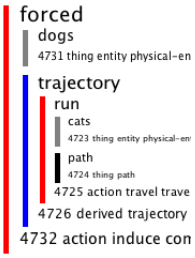


A `describe` is a simple representation for asking Genesis to describe something.
For example, `describe birds` corresponds to:



A `force` represents an agent forcing something to happen. It is a higher-order representation because that 'something' can be any other representation.

For example, `the dogs forced the cats to run` corresponds to:



A.2 Complete Training Examples for UNDERSTAND

This section includes the complete list of training examples used to create GENESIS' new semantic parser. The positive examples are very human-readable, because they were directly entered by a human. The negative examples, on the other hand, are much more verbose because they were entered via UNDERSTAND's graphical user interface.

All of the English is preceded by the tag `lp:` which indicates to UNDERSTAND that the Stanford link-parser should be used for the syntactic parsing.

The following text can be loaded as a file into UNDERSTAND to create the semantic parser described in this thesis:

```
Positive
lp: Bouvier is dog
(r "thing classification" (t bouvier) (t dog))

Positive
lp: in house
(d "derived pathFunction" (d in (d "thing at" (t house))))

Negative
lp: in the house
(D "derived pathfunction"
  (D "thing entity physical-entity object whole artifact structure housing dwelling house"
    (D "thing at" (T "thing part-of-speech part-of-speech-dt the"))))
(R "thing parse-link determiner"
  (T "thing entity physical-entity object whole artifact structure housing dwelling house")
  (T "thing part-of-speech part-of-speech-dt the"))
(T "thing entity physical-entity object whole artifact structure housing dwelling house")
(T "thing part-of-speech part-of-speech-dt the")

Positive
lp: bouvier is in house
```

```
(r "thing state" (t bouvier) (s "thing location" (* pathFunction)))

Negative
lp: bouvier ran in house
(R "thing state" (T "thing unknownWord bouvier") (S "thing location"
    (D "derived pathfunction" (D "thing part-of-speech part-of-speech-in in" (D "thing at"
        (T "thing entity physical-entity object whole artifact structure housing dwelling
            house"))))))
(T "action travel travel-rapidly run ran")

Positive
lp: above top of house
(d "derived pathFunction" (d above (d at (d top (t house)))))

Negative
lp: above top in house
(D "derived pathfunction" (D "thing part-of-speech part-of-speech-in above"
  (D "thing at" (D "thing entity physical-entity object location region top"
    (T "thing entity physical-entity object whole artifact structure housing dwelling house")))))
(T "thing part-of-speech part-of-speech-in in")

Positive
lp: boy ran
(d "derived trajectory" (r ran (t boy) (s "thing path")))

Negative
lp: boy is
(D "derived trajectory" (R "action be is"
  (T "thing entity physical-entity object whole living-thing organism person male male-child boy")
  (S "thing path")))
(T "action be is")

Negative
lp: boy ran to tree
(D "derived trajectory" (R "action travel travel-rapidly run ran"
  (T "thing part-of-speech part-of-speech-to to") (S "thing path")))
(R "thing parse-link prepositional-modifier" (T "action travel travel-rapidly run ran")
  (T "thing part-of-speech part-of-speech-to to"))
(T "thing part-of-speech part-of-speech-to to")

Negative
lp: boy ran to tree
(R "thing state"
  (T "thing entity physical-entity object whole living-thing organism person male male-child boy")
  (S "thing location" (D "derived pathfunction" (D "thing part-of-speech part-of-speech-to to"
    (D "thing at"
      (T "thing entity physical-entity object whole living-thing organism plant vascular-plant
          woody-plant tree"))))))
(T "action travel travel-rapidly run ran")

Positive
lp: boy ran to tree
(d "derived trajectory" (r ran (t boy) (s "thing path" (* pathFunction))))

Positive
lp: boy ran from house to tree
```

```
(d "derived trajectory"
  (r ran (t boy) (s "thing path" (d "derived pathFunction" (* from)) (* pathFunction))))

Positive
lp: vase fell off of shelf
(d "derived trajectory" (r fell (t vase) (s "thing path" (d off (* at)))))

Positive
lp: vase fell off shelf
lp: vase fell off of shelf

Negative
lp: the boy ran from the house to the tree
(D "derived trajectory" (R "action travel travel-rapidly run ran"
  (T "thing entity physical-entity object whole living-thing organism person male male-child boy")
  (S "thing path" (D "thing part-of-speech part-of-speech-to to" (D "thing at"
    (T "thing part-of-speech part-of-speech-in from"))))))
(R "thing parse-link prepositional-modifier" (T "action travel travel-rapidly run ran")
  (T "thing part-of-speech part-of-speech-to to"))

Positive
lp: bird appeared
(d "derived transition" (d appeared (t bird)))

Negative
lp: bird is
(D "derived transition" (D "action be is"
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      bird")))
(T "action be is")

Negative
lp: bird looks
(D "derived transition" (D "action look looks"
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      bird")))
(T "action look looks")

Positive
lp: bird disappeared
(d "derived transition" (d disappeared (t bird)))

Negative
lp: robin disappeared because kestrel flew over tree
(D "derived trajectory" (R "action travel fly flew"
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      bird bird-of-prey hawk falcon sparrow-hawk kestrel")
  (S "thing path" (D "thing part-of-speech part-of-speech-in because" (D "thing at"
    (T "thing entity physical-entity object whole living-thing organism plant vascular-plant
        woody-plant tree"))))))
(T "thing part-of-speech part-of-speech-in because")
(R "thing parse-link marker" (T "action travel fly flew")
  (T "thing part-of-speech part-of-speech-in because"))

Positive
lp: robin disappeared because kestrel flew
```

```
(r because (* trajectory) (* transition))

Negative
lp: robin disappeared because kestrel appeared
(R "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
    bird bird-of-prey hawk falcon sparrow-hawk kestrel"
  (D "derived transition" (D "action be look appear appeared"
    (T "thing part-of-speech part-of-speech-in because")))
  (D "derived transition" (D "action disappear disappeared"
    (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
        bird passerine oscine thrush robin"))))
(R "thing parse-link nominal-subject" (T "action be look appear appeared")
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      bird bird-of-prey hawk falcon sparrow-hawk kestrel"))
(T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
    bird bird-of-prey hawk falcon sparrow-hawk kestrel")

Negative
lp: robin disappeared because kestrel appeared
(D "derived transition" (D "action be look appear appeared"
  (T "thing part-of-speech part-of-speech-in because")))
(T "thing part-of-speech part-of-speech-in because")
(R "thing parse-link marker" (T "action be look appear appeared")
  (T "thing part-of-speech part-of-speech-in because"))

Positive
lp: dog jumped
(d "derived trajectory" (r jumped (t dog) (s "thing path")))

Positive
lp: did dog run
(d "derived question" (d did (* trajectory)))

Positive
lp: imagine dog jumped
(d imagine (* trajectory))

Positive
lp: describe birds
(d describe (t birds))

Negative
lp: elaborate the birds
(D "action act interact communicate inform explain clarify elaborate"
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      bird birds"))
(T "action act interact communicate inform explain clarify elaborate")

Negative
lp: did contact appear after the bird flew
(D "action be look appear" (D "derived trajectory" (R "action travel fly flew"
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      bird")
  (S "thing path"))))
(R "thing parse-link adverbial-clause-modifier" (T "action be look appear")
  (T "action travel fly flew"))
```

```
Positive
lp: dogs forced cats to run
lp: cats run

Negative
lp: dogs forced cats to run
(D "derived trajectory" (R "action travel travel-rapidly run"
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      mammal placental carnivore canine dog dogs")
  (S "thing path")))
(R "thing parse-link nominal-subject" (T "action induce compel coerce force forced")
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      mammal placental carnivore canine dog dogs"))

Positive
lp: dogs forced cats to run
(r forced (t dogs) (* trajectory))

Negative
lp: bird flew because dog appeared
(R "action be look appear appeared" (T "thing part-of-speech part-of-speech-in because")
  (D "derived question" (D "action be look appear appeared" (D "derived trajectory"
    (R "action travel fly flew"
      (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
          bird")
      (S "thing path"))))))
(T "thing part-of-speech part-of-speech-in because")
(T "action be look appear appeared")
(R "thing parse-link marker" (T "action be look appear appeared")
  (T "thing part-of-speech part-of-speech-in because"))

Negative
lp: bears forced the fish to go
(R "thing classification" (T "thing part-of-speech part-of-speech-dt the")
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      aquatic-vertebrate fish"))
(R "thing parse-link determiner"
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      aquatic-vertebrate fish")
  (T "thing part-of-speech part-of-speech-dt the"))
(T "action travel go")
(R "thing parse-link infinitival-modifier"
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      aquatic-vertebrate fish")
  (T "action travel go"))

Positive
lp: bears forced the fish to go
lp: fish go

Negative
lp: contact appeared between ball and block
(R "thing state"
  (T "thing entity abstraction psychological-feature event act action interaction contact")
  (S "thing location" (D "derived pathfunction" (D "thing part-of-speech part-of-speech-in between"
```

```
    (D "thing at"
      (T "thing entity physical-entity object whole artifact instrumentality equipment game-equipment
          ball"))))))
(T "action be look appear appeared")

Positive
lp: contact appeared between ball and block
(d "derived transition" (d appeared (r contact (t block) (t ball))))

Positive
lp: ball touched block
lp: contact appeared between ball and block

Negative
lp: ball passed block
(D "derived pathfunction" (D "thing part-of-speech part-of-speech-in between" (D "thing at"
  (T "thing entity physical-entity object whole artifact instrumentality equipment game-equipment
      ball"))))
(T "action travel pass passed")

Negative
lp: why did the ball touch the block
(D "derived pathfunction" (D "thing part-of-speech part-of-speech-in between" (D "thing at"
  (T "thing part-of-speech part-of-speech-wrb why"))))
(R "thing parse-link nominal-subject" (T "action touch")
  (T "thing entity physical-entity object whole artifact instrumentality equipment game-equipment
      ball"))
(T "thing part-of-speech part-of-speech-wrb why")
(R "thing parse-link adverbial-modifier" (T "action touch")
  (T "thing part-of-speech part-of-speech-wrb why"))

Negative
lp: man described
(D "action act interact communicate inform explain clarify elaborate set-forth describe described"
  (T "thing entity physical-entity object whole living-thing organism person male man"))
(R "thing parse-link nominal-subject"
  (T "action act interact communicate inform explain clarify elaborate set-forth describe described")
  (T "thing entity physical-entity object whole living-thing organism person male man"))

Negative
lp: bird flew to see
(D "derived question" (D "action perceive see" (D "derived trajectory" (R "action travel fly flew"
  (T "thing entity physical-entity object whole living-thing organism animal chordate vertebrate
      bird")
  (S "thing path")))))
(R "thing parse-link xclausal-complement" (T "action travel fly flew") (T "action perceive see"))
```

B Using UNDERSTAND in GENESIS

This appendix discusses the use of the new semantic parser infrastructure which I added to GENESIS. It shows how the UNDERSTAND program can be used to teach the semantic parser new examples, and how the resulting new rules can be incorporated into GENESIS.

B.1 How to Train UNDERSTAND

Currently, training the semantic parser is not done within the GENESIS system itself, but instead in a separate UNDERSTAND program.

In the Java software, the UNDERSTAND user interface is located in the executable class `understand.GUI`. See Figures 2 and 3 for screenshots of this program.

The original UNDERSTAND paper[4] is a good reference for how to train using positive and negative examples. The only changes I made to the user interface in this thesis are the addition of buttons for stepping the rule engine, as described in Section 3.1.2.

To augment the current set of training examples, choose `File>Open...` and select the file `understand/rules.txt`, which is the list of examples, as reproduced in Appendix A of this thesis.

We can then add new rules or refine old rules as described in the original UNDERSTAND paper. In doing so we should follow the training discipline detailed in Section 2.2 of this thesis.

Currently the UNDERSTAND interface has no way to delete examples, so if we realize we don't in fact want to teach an example we just provided, we need to save the rules and then manually delete the example. To do so, choose `File>Save`, use a text editor to remove the unwanted example from `understand/rules.txt`, and then reload the examples with `File>Open...`

Lastly, when the training is complete and we want to see how the newly trained semantic parser works within understand, we save the list of examples with `File>Save`.

We must *also* save a serialized copy of the semantic parse rules to the file `understand/rules.ser.gz` with

`File>Save Serialized...`. This is the file that GENESIS actually reads, so that it doesn't have to re-learn from the examples every time it starts up.

B.2 How to Use UNDERSTAND in GENESIS

Once `rules.ser.gz` has been updated as in the previous section, the changes will be reflected on the next launch of GENESIS' executable Java class, `piranha.Gauntlet`.

UNDERSTAND is not currently the default semantic parser in GENESIS, however, so it must be selected from the `Parser` menu. After this, any sentence processed by GENESIS will be sent through the updated semantic parser.

Currently, the output of `understand.Understand` is wired into a `piranha.UnderstandProcessor`, which in turn is wired into GENESIS (via a switchbox based on the selection in the `Parser` menu). This `UnderstandProcessor` does two things. Firstly, it throws out any leftover, 'raw' syntactic structures, to reduce clutter in the GENESIS interface. Secondly, this is where the thread modifications specified by any `derived` keywords (as described in Section 3.3) take place.

References

[1] John R. Bender. Connecting language and vision using a conceptual semantics. Master's thesis, MIT, 2001.

[2] Boris Katz, Gary Borchardt, and Sue Felshin. Natural language annotations for question answering. In *Proceedings of the 19th International FLAIRS Conference (FLAIRS 2006)*, 2006.

[3] Dan Klein and Christopher D. Manning. Fast exact inference with a factored model for natural language parsing. In *Advances in Neural Information Processing Systems 15 (NIPS 2002)*, pages 3–10. MIT Press, 2003.

[4] Michael T. Klein. Understanding english with lattice-learning. Master's thesis, MIT, 2008.

[5] Adam Kraft. Learning, using examples, to translate phrases and sentences to meanings. Master's thesis, MIT, 2007.

[6] George A. Miller. Wordnet 3.0, 2006. http://wordnet.princeton.edu/.

[7] Lucia Vaina and Richard Greenblatt. The use of thread memory in amnesic aphasia and concept learning, 1979.

[8] Patrick Winston. *Learning Structural Descriptions from Examples.* PhD thesis, MIT, 1970.