Using Near Misses to Teach Concepts to a Human Intelligence System

by

Jake A. Barnwell

B.S., Massachusetts Institute of Technology (2016)

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

February 2018

Author . . . .
Department of Electrical Engineering and Computer Science
January 18, 2018

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

Accepted by . . . .
Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Using Near Misses to Teach Concepts to a Human Intelligence System

by

Jake A. Barnwell

Submitted to the Department of Electrical Engineering and Computer Science
on January 18, 2018, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

If we are to understand how we can build artificial intelligence machines that are able to organically process and acquire knowledge like humans do, we must first understand how to teach a human intelligence system to model and learn generic concepts without resorting to mechanistic hand-coding of such concepts. Such an achievement in human intelligence is an important stride towards the realization of Minsky's hypothetical "baby machine."

Genesis is a story-understanding system that reads and analyzes stories in simple English, as humans do. Genesis can recognize certain sequences of events as particular manifestations of some user-defined concepts, such as *revenge* or *physical retaliation*. However, both the structure and definition of these high-level concepts must be explicitly provided by the user, and must be exactly matched using concept patterns against sequences of events in a story. I propose that this paradigm is unlike how humans learn such concepts, and instead, such concepts are learned by *example*, **not by explicit definition**.

To introduce this organic, example-driven paradigm to Genesis, I have created STUDENT, a system that ingests a small series of positive and negative examples of concepts and builds an internal model for these concepts. By aligning similar examples of concepts together, I have constructed refined concept patterns which encompass multiple, different manifestations of the concepts, without requiring a human to explicitly define them.

Adding this capability to Genesis enables it to learn concepts by example from a teacher. This behavior emulates how maturing humans learn concepts. Achieving this ability to emulate a human's learning process for a new idea brings us one step closer to modeling human intelligence and story understanding.

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

Acknowledgments

I would not be here at MIT, much less writing this thesis, without my parents' undying support. Even when they weren't sure what I was doing—even when I wasn't sure what I was doing—they believed in me.

I'm grateful for Rebecca's consistency and constancy. Her continuous questions, steady assurance, and occasional admonishment have kept me on track and in focus.

Thank you to Abdi, Jason, and Vincent, for putting up with my ridiculous jokes, and for providing a few of their own. For the late night discussions on pragmatics versus semantics. For their historical insights into the cultures of various regions of Italy.

Thank you to Dylan and Jessica, who have taught me so much about writing, learning, and teaching; and have always been ready to hear my ideas, complaints, and successes.

My infinite gratitude goes to Patrick, who has taught me that AI is much more than just algorithms and efficiency. You have pushed me to reach for the distant while also preserving the original spirit of my work. You have appropriated the principle of creative misunderstanding: in every meeting with you, I have learned things about my own research that neither of us knew in the first place. Without your guidance, I would still be sitting at a blank thesis proposal.

Contents

List of Figures

Chapter 1

Humans Learn by Example

If I asked you to write down an exact definition of «*assassination*», or «*revenge*», could you?

Maybe your definition of revenge would be something like:

> *revenge* $\implies$ X harms Y, so Y harms X (where X and Y are people)

This is a reasonable definition at first glance, but it lacks nuance. Some possible considerations are expounded below:

- ▷ If *X* says something mean about *Y*, is that considered "harm"? If so, we need to change our definition of «*revenge*» to qualify the scope of "harm".
- ▷ If *X* slaps *LY* (*Y*'s lover) and consequently *Y* slaps *X*, is this retaliation on *LY*'s behalf an example of revenge? If so, we need a way to incorporate relationship or common interest into our definition of «*revenge*».
- ▷ If *X* says something mean about *Y*, then *Y* kills *X*, can this be attributed to revenge? *Y*'s retaliation seems extreme, so perhaps this isn't «*revenge*», but instead just *Y* being insane. We need a way to recognize and forbid certain types of "harm".

You could modify your original definition of «*revenge*», generalizing and specifizing certain aspects of it to correctly account for each of the above situations. However, at what point does this become mechanistic instead of illustrative? For every definition you write, there's a more correct and nuanced definition that hasn't yet been written.

This dilemma suggests a case of "you know it when you see it": a phenomenon illustrating that, though you perhaps can't give an accurate definition of the concept, you can still supply, affirm, or refute an example of it.

For most people, this is not an uncommon phenomenon. Have you ever been asked to define the word "blue"? Or "and"? Or "expect"? It's quite difficult to give a good definition for any of these words. Sometimes, it is much easier for you to point to things that express the quality—

"That car over there is *blue*!"

or, use the word in a sentence to illustrate its syntactic behavior—

"That car is big *and* red."

or, give an example that highlights its semantic meaning—

"His car was much smaller than I *expected*."

In each of these situations, instead of providing an intensional description of the word's meaning, you have given an *ostensive definition* by providing an example of the word's usage—because it was much easier that way. I codify this idea as the **Principle of Ostension**.

> **Ostension**: It is often easier to demonstrate an idea with examples rather than define it explicitly.

Given how much humans communicate ideas to others via ostension, it only makes sense that the examples are well-received: that is, that the ideas are *learned* effectively from these ostensive definitions. Indeed, I claim that humans learn many things—words, actions, attributes, concepts—in an example-driven manner. This is clearly seen in young children. Young children do not learn...

▷ what a car is, or
▷ what friendship is, or
▷ what language is, or
▷ what blue is

...from a dictionary. Instead, young children learn these things by example, experience, and exploration. This corollary to the **Principle of Ostension** is formalized in what I call the **Principle of Exemplar**:

> **Exemplar**: Humans learn by experience rather than definition: ingesting examples is easier to internalize than reading a definition.

The **Principle of Exemplar** is partially inspired by Minsky's brief discussion of the *baby-machine*, a theoretical machine that emulates an infant who acquires knowledge of the world through experience (Minsky, 2006). Though Minsky is generally dismissive of our current ability to develop a full baby-machine, I believe that STUDENT, which is built with the **Principle of Exemplar** in mind, is a step in the right direction.

Unfortunately, it is not sufficient to motivate STUDENT's development by appealing to its ability to learn by example. After all, even deep neural networks learn by example—albeit thousands or millions of them. The primary appeal to the **Principle of Exemplar**, then, isn't the ability to learn from examples *ipso facto*: it is the ability to learn from a *small number* of examples. After all, human children don't learn new concepts by being fed millions of example sentences: they learn from just a few.

In any context of learning, there is always a *student* and a *teacher*. For example...

- ▷ The student might be a child learning about cars; or, a machine learning about stock trends; or, a neural network learning about categorizing images.
- ▷ The teacher might be a parent describing embarrassment; or, a professor explaining differential calculus; or, or a data scientist providing a database of images.

In any context of learning, care must be taken to select informative examples—though the definition of "informative" may depend on the situation. If a student is learning about Pyrrhic victory, the professor may select a passage from a book to demonstrate a case of Pyrrhic victory. On the other hand, if a neural network is learning what a dog is, the data scientist will likely provide many, many very similar pictures of (say) Golden Retrievers. In any case, all provided examples should be *accurate* and *discriminative*.

When there are just a few examples, the teacher must take extra care to select particularly informative examples. This is a perfectly acceptable compromise to not relying on massive amounts of data: after all, the objective of this research is not to pursue a system that can learn **without** an involved teacher—that would be ridiculous. The objective is instead to properly emulate how humans learn in real life: with a few, targeted examples. This objective is formalized in the **Principle of Quintessence**.

> **Quintessence**: Humans, and machines, can learn from a few, targeted examples chosen by a cognizant teacher.

If we want an artificial intelligence system to properly emulate the human thought process, it must be able to learn pursuant to the **Principle of Quintessence**. Winston (Winston, 1970) and Klein (Klein, 2008), described later in Section 2.2, developed algorithms that allowed us to embark on the long journey towards this goal, but I have leveraged their work to bridge the large gap between atomic *ideas* and full-scale *concepts*:

The Genesis System (described in Section 2.1) is a story-understanding system that can reason about concepts given their explicit definitions, but relies on the precision of the definitions supplied by users. I have created a new system, STUDENT, which imbues Genesis with the ability to learn concepts by a few, targeted examples provided by a teacher. STUDENT is a system that...

1. Formalizes a hierarchical representation for concepts that emphasizes semantic regularities among similar concepts;
2. Understands how to mutate one concept representation into another;
3. Quantifies the different types of mutations possible between similar concepts;
4. Reconciles differences between similar concepts and sentences;
5. Recognizes permissible sentences as manifestations of a concept, and rejects forbidden sentences that are not manifestations of a concept; and,
6. Explains its actions and thought process when making decisions.

Consider the task of teaching a computer the meaning of «*revenge*». STUDENT can quickly learn the meaning of «*revenge*» by analyzing a sequence of a few examples chosen by a teacher. The teacher starts by providing the following statement to STUDENT (as in Figure 1-1, (a)):

"Mary's harming Sue leads to Sue's harming Mary" is an example of «*revenge*».

From this example, STUDENT has already formed an initial model of what «*revenge*» might look like—albeit a quite primitive one. When the teacher asks

Is "Mary's harming John leads to John's harming Mary" an example of «*revenge*»?

STUDENT responds negatively (Figure 1-1, (b)). This is because STUDENT's current belief of «*revenge*» is exactly "Mary's harming Sue leads to Sue's harming Mary", as given in the first example.

In Figure 1-1 (c), the teacher declares that "Mary's harming John leads to John's harming Mary" is indeed an instance of «*revenge*». Because STUDENT has now seen two different people (Sue and John) as the retaliating party, STUDENT henceforth assumes that *any* person can fit in that role. So, in Figure 1-1 (d) and (e), STUDENT confirms that the candidates are both recognized as «*revenge*».

In Figure 1-1 (f), the teacher chooses an example that teaches STUDENT that the *instigator* can be any person; this is confirmed in (g).

When asked if

"Mary's harming Sue leads to Sue's **accidentally** harming Mary"

is an instance of «*revenge*» (Figure 1-1, (h)), STUDENT responds yes, even though as a human, you recognize that if Sue is **accidentally** harming Mary, it probably isn't «*revenge*». The teacher corrects this mistake (Figure 1-1, (i)), affirming to STUDENT that

"Mary's harming Sue leads to Sue's accidentally harming Mary"

is **not** an example of «*revenge*». Lastly, the teacher ensures that STUDENT has properly learned about "accidentally" by confirming that STUDENT does not consider

"Alice's harming Bob leads to Bob's accidentally harming Alice"

to be an instance of «*revenge*» (Figure 1-1, (j)).

From a sequence of just four examples, the teacher has taught STUDENT what it means to have revenge.

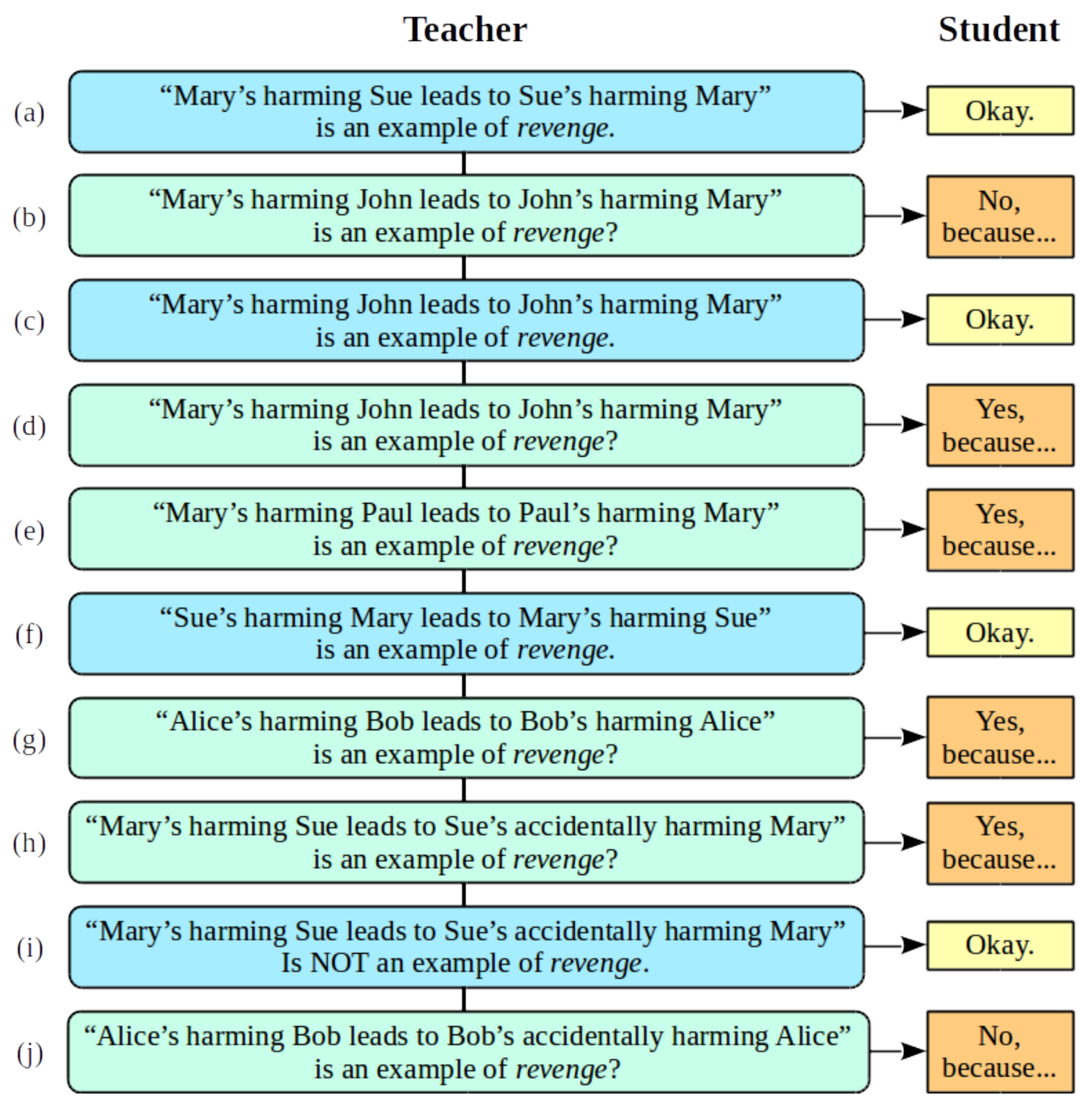

Teacher
Student
(a)
"Mary's harming Sue leads to Sue's harming Mary" is an example of *revenge*.
Okay.
(b)
"Mary's harming John leads to John's harming Mary" is an example of *revenge*?
No, because...
(c)
"Mary's harming John leads to John's harming Mary" is an example of *revenge*.
Okay.
(d)
"Mary's harming John leads to John's harming Mary" is an example of *revenge*?
Yes, because...
(e)
"Mary's harming Paul leads to Paul's harming Mary" is an example of *revenge*?
Yes, because...
(f)
"Sue's harming Mary leads to Mary's harming Sue" is an example of *revenge*.
Okay.
(g)
"Alice's harming Bob leads to Bob's harming Alice" is an example of *revenge*?
Yes, because...
(h)
"Mary's harming Sue leads to Sue's accidentally harming Mary" is an example of *revenge*?
Yes, because...
(i)
"Mary's harming Sue leads to Sue's accidentally harming Mary" Is NOT an example of *revenge*.
Okay.
(j)
"Alice's harming Bob leads to Bob's accidentally harming Alice" is an example of *revenge*?
No, because...

Figure 1-1: STUDENT can quickly learn a concept from just a few examples. In (b), (d), (e), (g), (h), and (j) above, the teacher asks STUDENT whether a particular sentence is an instance of «*revenge*»; STUDENT responds according to its current model of «*revenge*». In (a), (c), (f), and (i), the teacher actually provides examples to STUDENT to help it learn about «*revenge*».

Chapter 2

Background: Setting the Stage for Learning

In this chapter, I discuss past work that has formed the scaffolding underpinning STUDENT's success. In particular,

1. I introduce the Genesis System, a program that reads, understands, analyzes, and tells stories. I also discuss *Innerese*, Genesis' powerful internal story representation; STUDENT takes advantage of this representation to manipulate concepts.
2. I introduce near-miss learning, a genre of algorithms that enable systems to learn ideas from a series of successive examples provided by a knowledgeable teacher. Then, I survey several systems that use near-miss learning to teach different types of concepts and analyze story similarity.

2.1 Genesis is a substrate for story understanding

The Genesis system is an ongoing research project of the Genesis group at the MIT Computer Science and Artificial Intelligence Laboratory. The Genesis group holds that story understanding is the key to human intelligence. As such, the goal of the Genesis system is to create a human intelligence system that can read, understand, and generate stories in English, modeling the way we humans can do the same. To achieve this goal, Genesis must be able to do at least the following to a given story written in standard English:

1. Tokenize and parse the story into a semantically-unambiguous form.
2. Transform the semantically-unambiguous form into some internal data structure amenable to analysis; we call this internal representation *Innerese*.
3. "Read" the story via the Innerese structure and analyze it on the fly, applying rules Genesis knows in order to make inferences, connect plot points, make judgments, and explain events.
4. (For story generation) Transform Innerese into human-readable English; this is the inverse of the first two steps.

2.1.1 Parsing a story

Genesis uses the START parser (Katz, 1993) to transform English text into a semantically-unambiguous form; for each sentence, this form is a list of ternary expressions each of the form

```
(subject relation object)
```

representing all of the semantic data included in the sentence. For example, the sentence "John tenderly kisses Mary." yields two triplets [1]:

```
[(John kiss Mary),(kiss has_modifier tenderly)]
```

[1] Technically, the START parse of most sentences yields far more than two triplets. However, most indicate lexical relations such as `(kiss has_person 3)`, and are irrelevant for this discussion.

2.1.2 Transforming a story into Innerese

The semantically-unambiguous ternary representation is then converted into the Genesis Innerese representation.

An Innerese bundle is a collection of threads, each describing a word-sense

The linchpin of Innerese is the **bundle**, a mostly-unordered set of **threads** which, together, define the meaning of each word.

Genesis leverages Princeton's WordNet (Miller, 1995) to get the set of possible threads for a given token (word). Each thread is a semantic hierarchy of classes to which the particular word belongs. For example, two possible threads for the token "dog" are:

- ▷ `(thing entity [...]  carnivore canine dog)`
- ▷ `(thing entity [...]  person male chap dog)`

Each different thread embodies a different word-sense for the token. In the example above, the first thread for "dog" refers to the animal, and the second refers to the slang term for a male human.

Many common words have far more than just two threads. For example, the word "staff" has many threads, as illustrated in Figure 2-1.

The bundle is unordered except for a single, primary thread, called the *primed thread*. The primed thread is the one that is often used during computation and analysis, as it represents the most likely word-sense of the token. For example, the primed thread for "dog" is often

```
(thing entity [...]  carnivore canine dog)
```

because typically, in human speech and language, the word "dog" is meant to refer to the animal.

Innerese comprises four types of recursively-defined objects

An Innerese expression is comprised of four types of Things, each defined recursively:

- ▷ `Entity (ent)`: Any atomic object, value, or idea, such as `laptop`, `red`, or `happiness`.

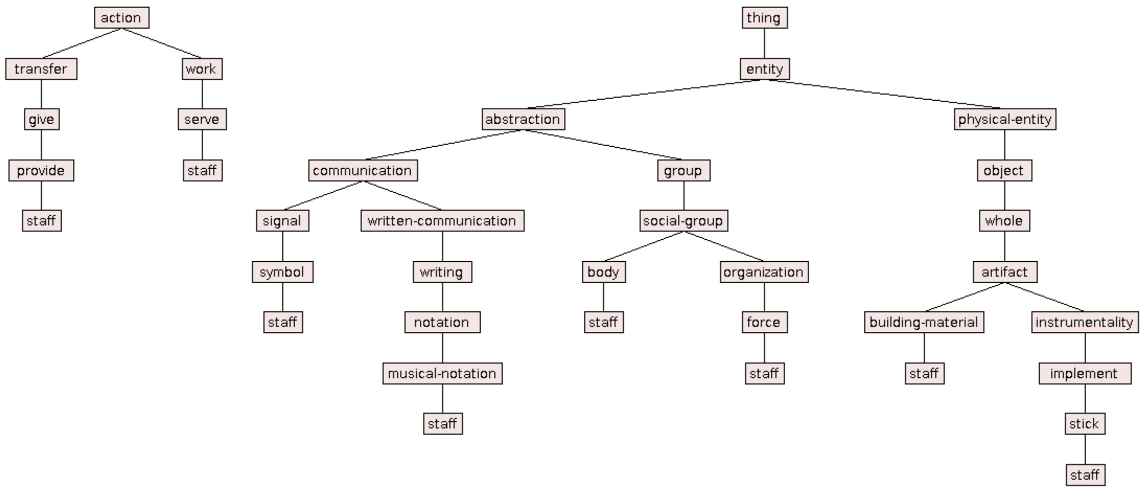


Figure 2-1: Some words can have many associated threads. The word "staff" has eight threads—eight different word-senses—associated with it. Two refer to actions (verbs), and six refer to things (nouns). Some words can have up to ten or more threads ("release"), while others might have just one thread ("however").

- ▷ `Function` (`fun`): Suggests something derived from a Thing, such as `top of the table` or `with a knife`.
- ▷ `Relation` (`rel`): Describes a connection or relationship between two different Things, such as `door beside window` or `John kisses Mary`.
- ▷ `Sequence` (`seq`): An ordered list of any number of Things.

For example, the Innerese form of the English sentence "John tenderly kisses Mary." is:

```
(rel kiss (ent john)
          (seq roles (fun object (ent mary))
                     (fun manner (ent tenderly))))
```

Each of the four types of Things has its own dedicated Java class, and all are under the umbrella of `Entity`, as summarized in Figure 2-2. In particular,

- ▷ An `Entity` represents an atomic idea defined by a **bundle** of **threads** stored within the object.
- ▷ A `Function` is an `Entity` with an added **subject** field.
- ▷ A `Relation` is a `Function` with an added **object** field.

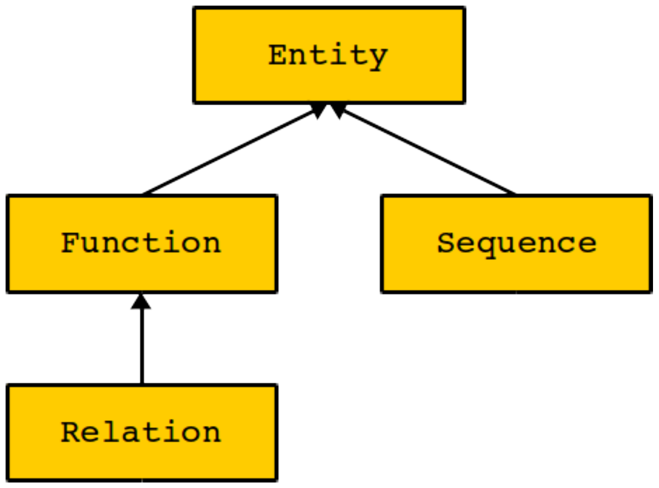


Figure 2-2: Innerese statements comprise four types of Things: `Entity`, `Function`, `Relation`, and `Sequence`. `Function`, `Relation`, and `Sequence` are all Java subclasses of `Entity`, expressing various semantic information stored in addition to that inherited from `Entity`.

▷ A `Sequence` is an `Entity` with an added **elements** field, enumerating all components that comprise the `Sequence`.

Raw `Entitys` often describe nouns (`(ent dog)`), adjectives (`(ent tall)`, and adverbs (`(ent tenderly)`).

A `Function` typically has one of three purposes:

▷ Indicate the direct object of verb, e.g. `(fun object (ent mary))`

▷ Signal a prepositional phrase, e.g. `(fun with (ent knife))`

▷ Designate an adverb ("manner"), e.g. `(fun manner (ent carefully))`

The single argument to a `Function` is called the *subject* of the `Function` (even though the subject can itself refer to a grammatical object!).

A `Relation` can be used to signal many things, including:

▷ A verb, e.g. `(rel kiss (ent john) (seq roles ...))`

▷ Entailment (causality), e.g. `(rel entail (rel ...) (rel ...))`

▷ Possession (ownership), e.g. `(own (ent john) (ent car))`

The two arguments to a `Relation` are called the *subject* and *object*.

A `Sequence` is used to structurally group multiple things in an ordered sequence, though the order may not be semantically important. One common application for a

`Sequence` is to encapsulate “roles” information from a verb, including the direct objects, indirect objects, prepositional phrases, and adverbs. For example, the English sentence

"John carefully slices the cake on the table."

translates to

```
(rel slice (ent john)
           (seq roles (fun object (ent cake))
                      (fun on (ent table))
                      (fun manner (ent carefully))))
```

in Innerese. However, `Sequence`s also have a few other uses:

- ▷ Technically, any Innerese translation is enveloped in a `Sequence` of type “semantic-interpretation”, indicating that the Innerese to follow is a semantic interpretation of a sentence, not necessarily a faithful syntactic snapshot.
- ▷ Conjunctions (“and”) and disjunctions (“or”) use a `Sequence` of type “conjunction” or “disjunction”, respectively, to enumerate each of the constituents.

The constituent items in the `Sequence` are referred to as the `Sequence`’s *elements*.

Figure 2-3 gives brief visual examples of the four types of Things.

2.1.3 Analyzing a story

Finally, Genesis can invoke common-sense and user-defined rules to find connections and infer plot elements in a story. For example, suppose the following rule is provided to Genesis:

```
If X kills Y, then Y becomes dead.
```

Then, if Genesis finds the sentence “John kills Sam.” in a story, it matches and applies the rule, concluding that “Sam becomes dead.”

Higher-level *concepts* can also be defined for Genesis, not just “common-sense” rules as above. Below, we explicitly define the concept of «*revenge*» for Genesis:

```
Start description of "revenge".
X is an entity.
```

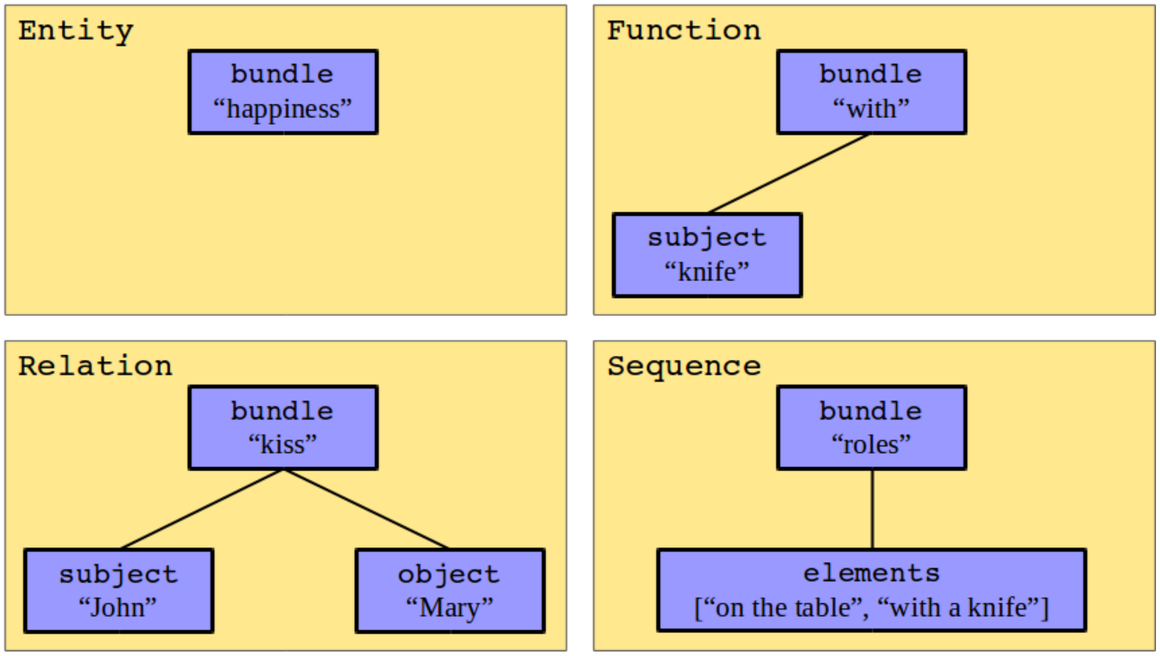


Figure 2-3: All Things are `Entitys`, each with different relevant information. `Entity` objects are purely defined by their **bundle**. `Functions` also contain a **subject**, and `Relations` have both a **subject** and an **object**. `Sequences` have neither a **subject** nor an **object**; instead, they have **elements**, an ordered list of constituents.

```
    Y is an entity.
    X's harming Y entails Y's harming X.
    The end.
```

Genesis can match this pattern to elements in a story. As such, if a story included the following sentence,

```
John's harming Sam leads to Sam's harming John.
```

then Genesis would recognize this occurrence as an example of «*revenge*».

2.2 Computers can learn from near misses

Winston successfully demonstrated the **Principle of Quintessence** in teaching a computer system to properly recognizes basic structures such as houses, tables, and arches (Winston, 1970). To teach his system about a structure, he provides several successive examples to the system; each teaches something new. Each example has two characteristics:

- ▷ The example is carefully selected by a teacher.
- ▷ The example is either (a) an instance of the structure being learned, or (b) a non-instance of the structure being learned.

Importantly, if the example is a non-instance of the structure being learned, it is a "near miss": it doesn't align with the system's current belief of the structure's description, but it only differs in at most a few significant ways.[2]

Because each subsequent example *almost* aligns with the system's current model, the system can easily infer the important information that is presented in the example. So, the system can learn something new at *every step*.

Winston's near-miss learning paradigm has paved the way for several artificial intelligence systems (introduced below) that benefit from teaching concepts requiring only a few examples, so-called "small data"—in contrast to big-data systems which require thousands or millions of examples.

I will briefly discuss some more recent work which builds on the principle of near misses. When discussing near-miss learning systems in this section and in the remainder of this thesis, I will often refer more generally to +positive and –negative examples:

- ▷ +Positive examples loosen restrictions on the system, teaching it what it is allowed to do.
- ▷ –Negative examples enforce restrictions on the system, preventing it from doing things it is not allowed to do.

[2]On the other hand, if the example **is** an instance of the structure being learned, and the model doesn't currently recognize it, the example is permitted to have any number of differences from the model.

2.2.1 Klein introduces lattice-learning to learn characteristics of entities

Some work with the Genesis system has used portions of Klein's lattice-learning concept (Klein, 2008), which is itself an application of near-miss learning. Klein uses lattice-learning in his UNDERSTAND system, which can learn to recognize characteristics of certain entities: for example, given just a few examples, UNDERSTAND can learn that birds and aircraft can all *fly*, while other things like insects and boats cannot. Lattice-learning can work with any kind of classification tree: by considering positive and negative examples of some feature, with each example located at some position in the classification tree, the system can *permit* entire regions of samples in the tree using nearby positive examples, and the system can *exclude* entire branches from the tree based on nearby negative examples. The classification trees used by Klein are informed by word-sense threads (discussed in Section 2.1.2) supplied by Princeton's WordNet (Miller, 1995).

2.2.2 Stickgold uses lattice-learning to disambiguate word-senses

Stickgold's DISAMBIGUATE (Stickgold, 2011) applies the methods of lattice-learning to finding similarities between multiple sentences. Once the similarity *context* is found, DISAMBIGUATE can infer the semantically appropriate meanings from possibly-ambiguous words, such as the noun "hawk" which may refer to either a bird or a malicious person.

2.2.3 Krakauer aligns multiple stories by finding similar concepts within

Krakauer attempts to compare multiple stories by finding similar templates or *concept patterns* among stories that seem to match to the same high-level concept, such as «*revenge*» (Krakauer, 2012). However, Krakauer approaches this problem with the goal of assessing story similarity, not with the goal of teaching Genesis semantic concepts. As such, the system is designed to recognize structural similarity, not to learn incrementally from various examples.

Chapter 3

Pedagogy: Philosophy of Learning

Built into any artificial intelligence program, whether implicitly or explicitly, is a set of assumptions that simplify the system or improve its efficiency. STUDENT is no different. However, as a system that incorporates aspects of human communication, STUDENT can take advantage of several rules that largely define how humans interact with each other:

▷ When humans communicate, they unconsciously abide by a set of principles restricting what can be said and how it should be said. These principles, or *maxims*, enable each person to make reasonably accurate assumptions about the pragmatics of the statements at hand.

▷ Teachers and students implicitly agree to follow a set of guidelines when in a learning environment: these guidelines, or *tenets*, facilitate the learning process.

3.1 Humans adhere to four maxims of communication

One of my favorite short jokes goes like this:

> A dad is washing the car with his son. After a moment, the son asks his father, "Do you think we could use a sponge instead?"

Some would say that if a joke needs to be explained, it isn't funny—but for the sake of leveling the playing field, I will explain. In the joke, the word-sense of "with" is ambiguous, as the object of the preposition ("his son") could either refer to a co-agent (the father and the son are both washing the car) or an instrument (the father uses the son like a sponge, to wash the car). Naturally, most people reading the joke assume the child's role is the co-agent, until the punchline reveals that the child is an instrument.

Consider also this rather humorous interaction:

> Ada and her husband John were watching TV one evening. Suddenly, Ada rose and announced that she was retiring to the bedroom.
>
> "Are you going to read or sleep?" John inquired.
>
> "Yes," Ada responded.

The punchline here isn't about lexical ambiguity; it's just that Ada answers the question logically ("yes" or "no" to a logical `OR`) instead of pragmatically ("which one?").

Both of these jokes poke fun at how humans communicate:

- ▷ In the first, the reader assumes a particular word-sense given that a fairly common English sentence is presented, but then the following sentence subverts that expectation.
- ▷ In the second, a fairly common "either–or" question is asked, but the recipient chooses to answer in a way that is *technically* correct but semantically meaningless.

These jokes both demonstrate violations of the unspoken rules by which humans typically communicate. Like many principles in the field of artificial intelligence, these rules are all but obvious once codified, but they give us powerful ways to reason about human intelligence and learning systems. These rules are formalized by Paul Grice and commonly known as Grice's Maxims (Grice, 1975):

1. **Maxim of Quantity**: Be informative. Give as much information as necessary, but no more.

2. **Maxim of Quality**: Be truthful. Say true things; don't mislead or lie.
3. **Maxim of Relation**: Be relevant. Speak about topics at hand, not irrelevant things.
4. **Maxim of Manner**: Be clear. Don't be vague or ambiguous.

Together, these four maxims describe the implicit contract by which we tend to communicate. For example, when a person is asked a question, she typically responds

- ▷ with sufficient information;
- ▷ accurately;
- ▷ pertinently; and
- ▷ clearly.

STUDENT takes advantage of this contract—Grice's Maxims—to enable it to learn relevant information from each example.

3.2 Teachers and students interact according to a covenant

STUDENT takes advantage of Grice's Maxims when making decisions about how to model concepts. However, because STUDENT learns by ingesting examples that are *carefully chosen* by a **teacher**, STUDENT can also take advantage of an additional contract: a set of tenets that describe how students and teachers communicate. These tenets are largely based on what VanLehn calls the felicity conditions for communication (Vanlehn, 1983). I will refer to them en masse as the student–teacher covenant.

The student–teacher covenant describes how a student and teacher should interact, with the assumption that *the student wants to learn something*, and *the teacher wants the student to learn something*. Many of the tenets implicitly rely on Grice's Maxims, but they are all more specified to the context of learning. The tenets of the covenant that I consider important to STUDENT's success are explained below.

▷ **The teacher only teaches things that the student already almost knows.** A calculus teacher introduces integrals not by giving a comprehensive sermon on different strategies for computing integrals, but instead by motivating the intuition for integrals via a topic with which the student is already familiar: visualizing areas under curves.

"What is the area under this parabola?"

When the student realizes she has no tool (equation) in her toolbox to compute the area, the teacher continues by approximating the area under the parabola as a series of many thin rectangles—a tedious, yet fundamentally calculable number.

"What is the combined area of these twenty rectangles?"

Figure 3-1 demonstrates a humorous, if extreme, non-example of this tenet.

▷ **The teacher gives information that is accurate and relevant.** The information taught by the teacher should obviously be accurate and not misleading in any way. Supplying inaccurate information only impedes progress. Furthermore, information given should be relevant to the topic at hand: off-topic or unrelated information, even if technically accurate, does not help the student learn the material at hand.

- ▷ **The teacher gives examples that allow the students to learn.** If a teacher provides examples to strengthen the student's comfort with the material, each example should provide useful information to the student. For example, no two examples should be the same, and no two examples should be contradictory.
- ▷ **The teacher does not begin with edge-case or intentionally-confusing examples.** In learning a new topic, basic but general knowledge attained is far more valuable than nuanced but mostly-unapplicable knowledge. For example, the following example

  "The area of this rectangle is 15 because its width is 3 and its height is 5."

  may not impart much generalizable knowledge on the student, but it is still likely more helpful to the student than

  "The area of this line segment is 0 because it's technically a rectangle with width 0."

  even though the later example contains useful, nuanced information that may prove useful sometime in the future.
- ▷ **The teacher does not choose examples that coincidentally express irrelevant characteristics.** For example, when describing a *rectangle*, the teacher should not choose a square as the first example:

  "This is a rectangle. Its width is 2 and its height is 2."

  The student may ascribe relevance where no relevance is due, in this case, to the fact that the height and width "must be equal," which is decidedly false for a rectangle. Similarly, if trying to teach about the color "red," the teacher should certainly not, as an example, write

  RED

  on the board using a red marker; such an example may cause the student to conflate the orthographic symbol RED with the color itself.
- ▷ **The student applies past knowledge to current topics.** Knowledge does not exist in a vacuum, and neither does the student. The student should use past knowledge to help her understand new concepts.
- ▷ **The student recognizes that concepts learned can be generalized.** The student does not assume that the material conforms exactly to the information provided by

the teacher. For example, given two worked examples

"The area of this 4 by 2 rectangle is 8."

"The area of this 3 by 7 rectangle is 21."

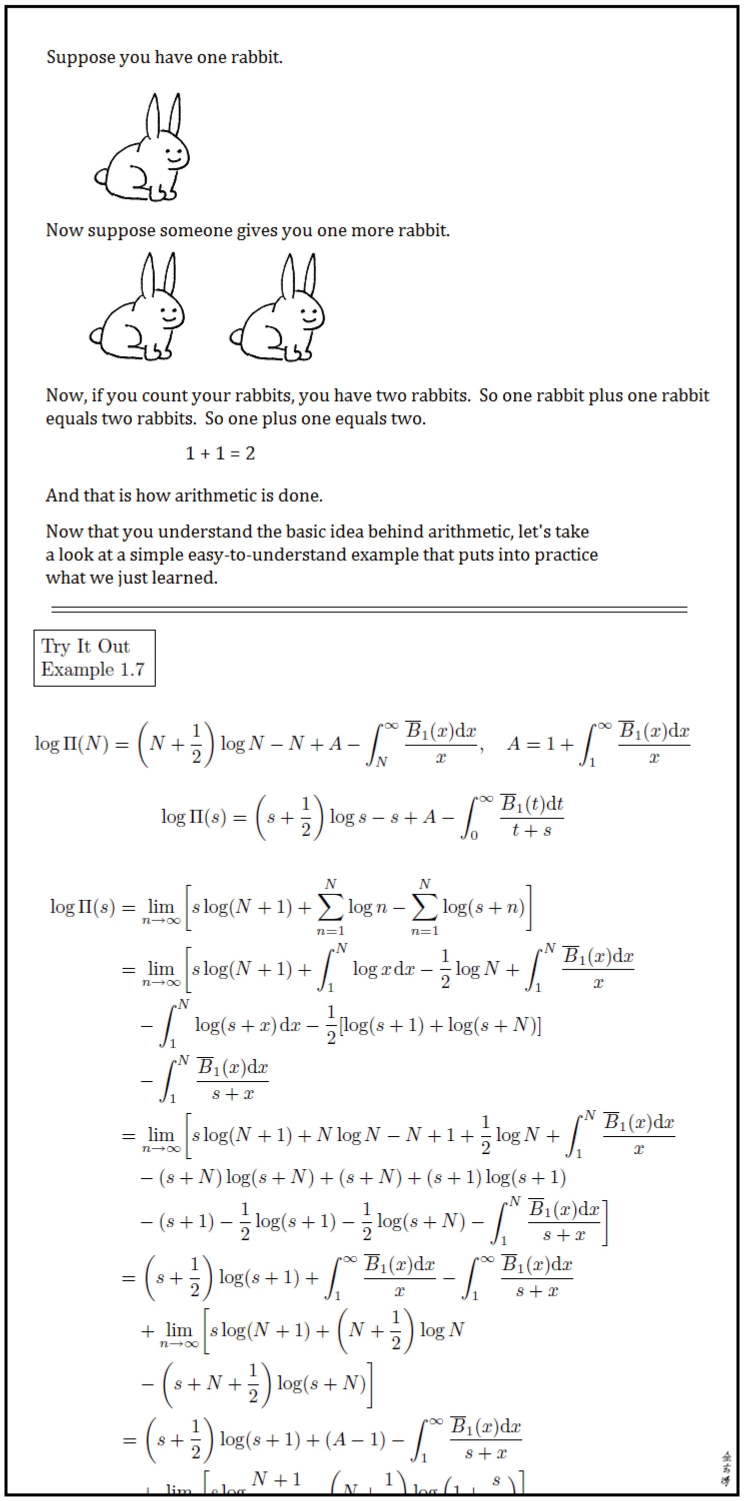
the student should not assume that only those two rectangles can have their area computed. (Perhaps the student can even assume that there is a pattern to how the area is computed...)

- ▷ **The student assumes that interesting characteristics expressed by examples are relevant.** If the teacher provides an example that expresses a non-trivial characteristic, the student should assume there is a reason the teacher chose that example. For example, in learning what a square is, if the teacher provides

  "This is a square. Its width is 2 and its height is 2."

  as an example, the student should ascribe some relevance to the fact that the width and height are the same.
- ▷ **The student explains her thought process while reasoning.** When thinking about a concept, or trying to solve a problem, the student should give feedback to the teacher in some manner, describing what she is thinking. If the student adheres to this tenant, then if the student comes to a wrong conclusion, the teacher will realize why. Or, if the student comes to the right conclusion, but for the wrong reason, the teacher will realize why.

When reasoning about concepts, STUDENT takes advantage of all of these tenets. This allows it to make more accurate logical deductions, and learn about concepts more efficiently. However, it also puts STUDENT at the mercy of the teacher, who must follow the covenant; if the teacher violates the covenant, STUDENT is likely to fail in creating an accurate model of the concept.



Suppose you have one rabbit.

Now suppose someone gives you one more rabbit.

Now, if you count your rabbits, you have two rabbits. So one rabbit plus one rabbit equals two rabbits. So one plus one equals two.

1 + 1 = 2

And that is how arithmetic is done.

Now that you understand the basic idea behind arithmetic, let's take a look at a simple easy-to-understand example that puts into practice what we just learned.

Try It Out
Example 1.7

$$\log \Pi(N) = \left(N + \frac{1}{2}\right)\log N - N + A - \int_N^\infty \frac{\overline{B}_1(x)\mathrm{d}x}{x}, \quad A = 1 + \int_1^\infty \frac{\overline{B}_1(x)\mathrm{d}x}{x}$$

$$\log \Pi(s) = \left(s + \frac{1}{2}\right)\log s - s + A - \int_0^\infty \frac{\overline{B}_1(t)\mathrm{d}t}{t+s}$$

$$\begin{aligned}
\log \Pi(s) &= \lim_{n\to\infty}\left[s\log(N+1) + \sum_{n=1}^{N}\log n - \sum_{n=1}^{N}\log(s+n)\right] \\
&= \lim_{n\to\infty}\Bigg[s\log(N+1) + \int_1^N \log x\,\mathrm{d}x - \frac{1}{2}\log N + \int_1^N \frac{\overline{B}_1(x)\mathrm{d}x}{x} \\
&\quad - \int_1^N \log(s+x)\,\mathrm{d}x - \frac{1}{2}[\log(s+1) + \log(s+N)] \\
&\quad - \int_1^N \frac{\overline{B}_1(x)\mathrm{d}x}{s+x} \\
&= \lim_{n\to\infty}\Bigg[s\log(N+1) + N\log N - N + 1 + \frac{1}{2}\log N + \int_1^N \frac{\overline{B}_1(x)\mathrm{d}x}{x} \\
&\quad - (s+N)\log(s+N) + (s+N) + (s+1)\log(s+1) \\
&\quad - (s+1) - \frac{1}{2}\log(s+1) - \frac{1}{2}\log(s+N) - \int_1^N \frac{\overline{B}_1(x)\mathrm{d}x}{s+x}\Bigg] \\
&= \left(s + \frac{1}{2}\right)\log(s+1) + \int_1^\infty \frac{\overline{B}_1(x)\mathrm{d}x}{x} - \int_1^\infty \frac{\overline{B}_1(x)\mathrm{d}x}{s+x} \\
&\quad + \lim_{n\to\infty}\Bigg[s\log(N+1) + \left(N + \frac{1}{2}\right)\log N \\
&\quad - \left(s + N + \frac{1}{2}\right)\log(s+N)\Bigg] \\
&= \left(s + \frac{1}{2}\right)\log(s+1) + (A-1) - \int_1^\infty \frac{\overline{B}_1(x)\mathrm{d}x}{s+x}
\end{aligned}$$

If the authors of computer programming books wrote arithmetic textbooks...

Figure 3-1: This comic from Abstruse Goose (AbstruseGoose, 2012) demonstrates an extreme violation of the student–teacher covenant.

Chapter 4

STUDENT: A Program that Learns Concepts from Near Misses

STUDENT is a human intelligence program—so called because it seeks to emulate human behavior—that can learn about concepts from a few near misses provided by a cognizant teacher. At its core, it uses the paradigm of near-miss learning to generalize and specifize particular aspects of its current belief of a concept, while also making decisions consistent with Grice's Maxims and the student–teacher covenant. STUDENT is built on top of the Genesis System, using Genesis' Innerese as an intermediate knowledge-representation to facilitate concept instantiation and manipulation.

In this chapter, you will learn

▷ How the teacher and the client interact with STUDENT;

▷ How STUDENT represents concepts internally;

▷ How concepts are instantiated and manipulated;

▷ How near misses are determined; and

▷ How new examples of concepts are reconciled.

4.1 STUDENT continuously updates its model of a concept from provided examples

STUDENT interacts with users in the manner of an anytime algorithm. When a client queries with a candidate for a concept, STUDENT will respond indicating whether it believes the candidate embodies the concept, and also explaining how it reaches that conclusion; but its concept model becomes more accurate as it learns more examples from a teacher.

The teacher intentionally chooses examples that allow STUDENT to most effectively learn from a small number of trials. Each example chosen is a *near miss*, an example that just barely contradicts STUDENT's current understanding of the concept's model, forcing STUDENT to learn from every single example.

In accordance with the covenant, STUDENT also gives feedback to the teacher, explaining its thought process as STUDENT ingests new examples and tries to reconcile near misses with the current belief state.

The interactions among STUDENT, the teacher, and the user are illustrated in Figure 4-1.

To create an initial model for a concept, STUDENT constructs it from a ♣seed example carefully chosen by the teacher in accordance with the **Principle of Relevancy**. This seed is directly translated into the STUDENT's inner representation, and is assumed at that point in time to exactly embody the concept in question. For example, to initialize the model for «*harm*», the teacher may choose to start by supplying

♣`John harms Mary.`

as the seed example. Henceforth, STUDENT assumes that «*harm*» is *exactly* defined as

> *harm* $\Longrightarrow$ John harms Mary.

until it receives further examples from the teacher.

The teacher may choose to supply any number of additional examples to further fine-tune STUDENT's belief of the concept. After the seed, every subsequent example ξ has associated with it a *charge*, which is either

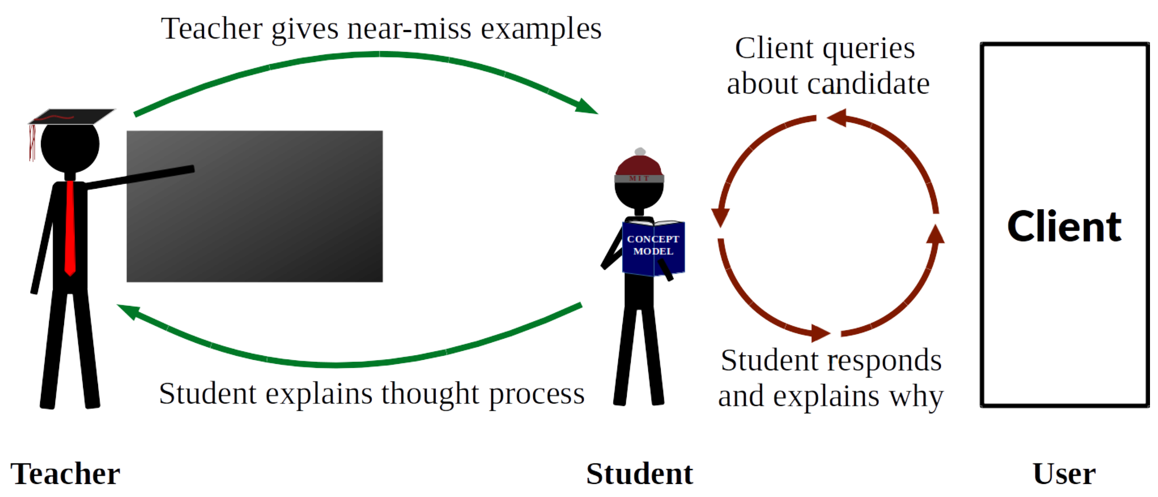

Teacher gives near-miss examples
Client queries about candidate
CONCEPT MODEL
Client
Student explains thought process
Student responds and explains why
Teacher
Student
User

Figure 4-1: STUDENT learns from the teacher, and can respond to the user's queries. The teacher intentionally chooses near-miss examples to guide STUDENT's creation of the concept model; STUDENT explains itself as it reconciles new examples with the model. The user can ask STUDENT if a particular candidate is an instance of a concept; STUDENT responds indicating yes or no, additionally explaining how it came to that conclusion.

▷ +positive, indicating that this example should be permitted (recognized) by the model as an instance of the concept; or,

▷ –negative, indicating that this example should be rejected by the model: it should not be considered an instance of the concept.

The new example ξ may have a charge that is consistent with the current model's prediction for ξ, described by one of the following two situations:

▷ ξ is a +positive example and STUDENT's concept model permits ξ.

▷ ξ is a –negative example and STUDENT's concept model rejects ξ.

In either case, ξ does not give STUDENT any actionable information, because ξ is already fully consistent with the model.

On the other hand, if ξ's charge is **opposite** of the model's prediction for ξ, this indicates a possible opportunity to learn new information—and to reconcile STUDENT's belief with that new information. However, in order to learn something new from an example, STUDENT requires that ξ is a **near miss**: **ξ must be nearly consistent with the model, so that it is obvious exactly how ξ can be reconciled with the model**. In particular,

- ▷ if ξ is a +positive example: STUDENT's model must just barely reject ξ—there must be exactly one change that can be made to the model to cause it to permit ξ.
- ▷ if ξ is a –negative example: STUDENT's model must just barely permit ξ—there must be exactly one change that can be made to the model to cause it to reject ξ.

If ξ is not a near miss, then STUDENT fails to learn anything, and reports such to the teacher.

This definition of near miss is a bit different from that used by Winston. Winston's program can learn from *any* +positive example, generalizing along *all* differing aspects, while it can only learn from –negative examples that are near misses. STUDENT, on the other hand, can only learn from a near miss *in either case*. I believe that enforcing this restriction on STUDENT more accurately emulates how teachers provide examples to students, only changing one primary aspect of the example at a time.

Once ξ has been confirmed to be a near miss, STUDENT can incorporate ξ into its belief, updating the model for the concept. If ξ is a +positive example, then STUDENT generalizes its model, permitting more varieties of candidates as instances of the concept. If ξ is a –negative example, then STUDENT specifizes its model, forbidding formerly-permissible examples in its model-space and pruning the space of permissible candidates.

STUDENT's general control flow is summarized in Figure 4-2.

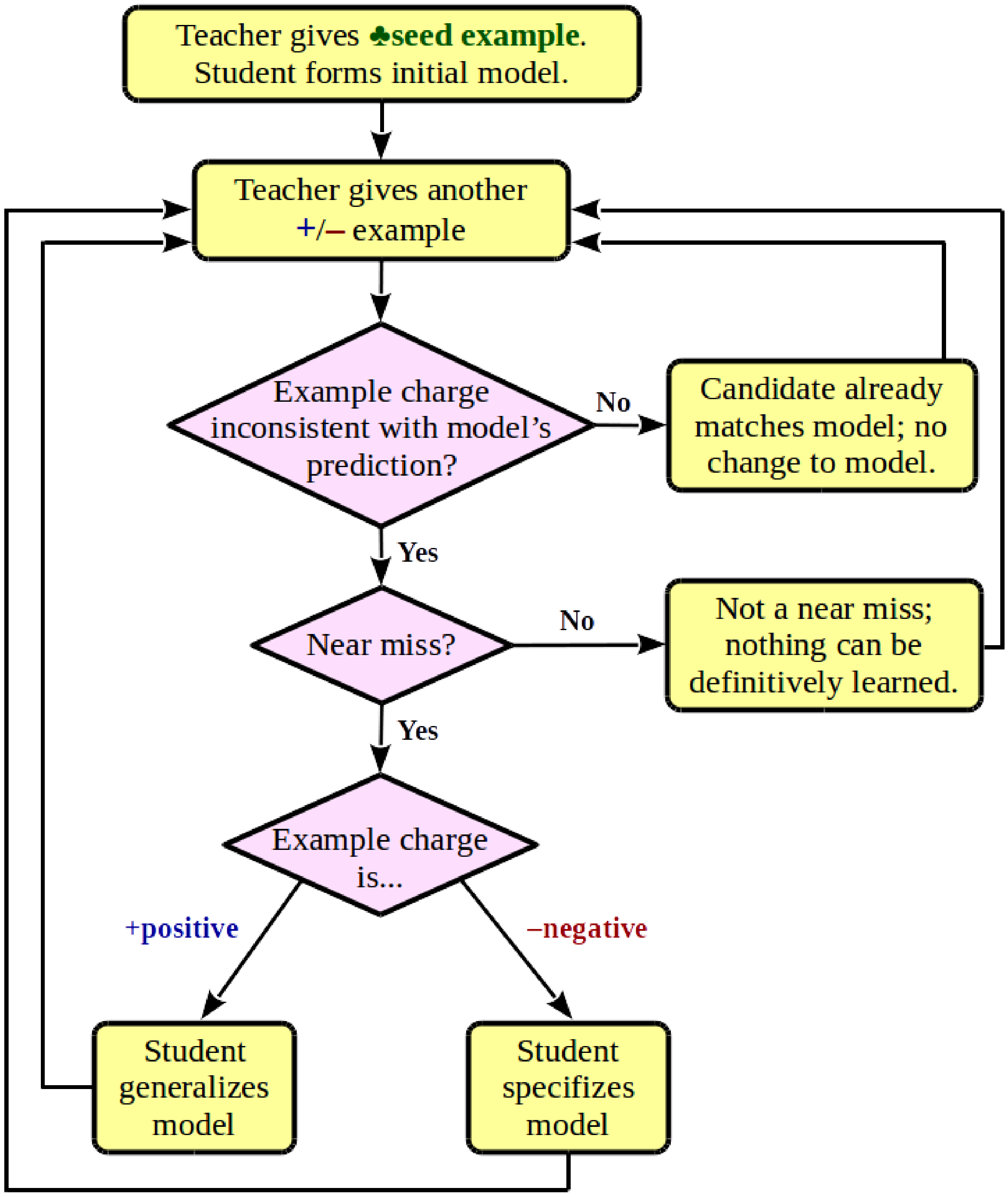

Teacher gives ♣seed example.
Student forms initial model.
Teacher gives another +/– example
Example charge inconsistent with model's prediction?
No
Candidate already matches model; no change to model.
Yes
Near miss?
No
Not a near miss; nothing can be definitively learned.
Yes
Example charge is...
+positive
–negative
Student generalizes model
Student specifizes model

Figure 4-2: STUDENT initializes its belief (model) of a concept from a ♣seed example first supplied by the teacher. Subsequent examples given by the teacher must be near misses (if they are not, no action can be taken to update the model). Such examples can either be +positive or –negative near misses. A +positive near miss is one that is not permitted by the current model, but the teacher claims it should. A –negative near miss is one that is permitted by the current model, but the teacher claims it should be forbidden.

4.2 A concept model is a semantic tree with local and global constraints

Genesis' inner language, Innerese, represents all Things as instances of `Entity`, or some subclass thereof (Section 2.1.2). A story is hence fully-defined by a sequence of `Entitys`, each one representing (roughly) a sentence.

Because all `Entity`-like objects are recursively defined, it is only reasonable that a concept's model, which seeks to embody the required structure and content of permissible `Entitys`, is also hierarchical in structure. Therefore, the vast majority of the information describing a concept is stored in a class called `ConceptTree`, which is a recursively-defined tree that is explained in much more detail in Section 4.3.3.

Because a `ConceptTree` is a recursively-defined tree, each node, corresponding to an `Entity`, can only encode *local* information relevant to the node (and perhaps its children and parent). This structure format is sufficient to describe node-specific conditions that constrain things like the word (thread) and type of children, but isn't well-suited to describe *global* information that constrains nodes in relation to other nodes, such as "nodes must-be-same" or "nodes must-be-different". For this reason, an encapsulating class called `ConceptAuthoritree` is used to package the rooted `ConceptTree`; it administers the global constraints—also referred to as structural constraints—that externally constrain sets of different nodes. The `ConceptAuthoritree` is the authoritative record of the full concept's model.

As shown in Figure 4-3, STUDENT stores the entire reasoning system, including the `ConceptAuthoritree`, in a class surprisingly called `ConceptModel`. The `ConceptModel`:

▷ Stores the record of the concept itself (the `ConceptAuthoritree`).

▷ Facilitates interaction between the client and the model.

▷ Maintains the full history of examples from the teacher.

The reason that the `ConceptModel` maintains a history of examples is so that the teacher can inquire about misunderstandings stemming from previous examples if necessary. Furthermore, future plans for STUDENT include the ability to reconcile past

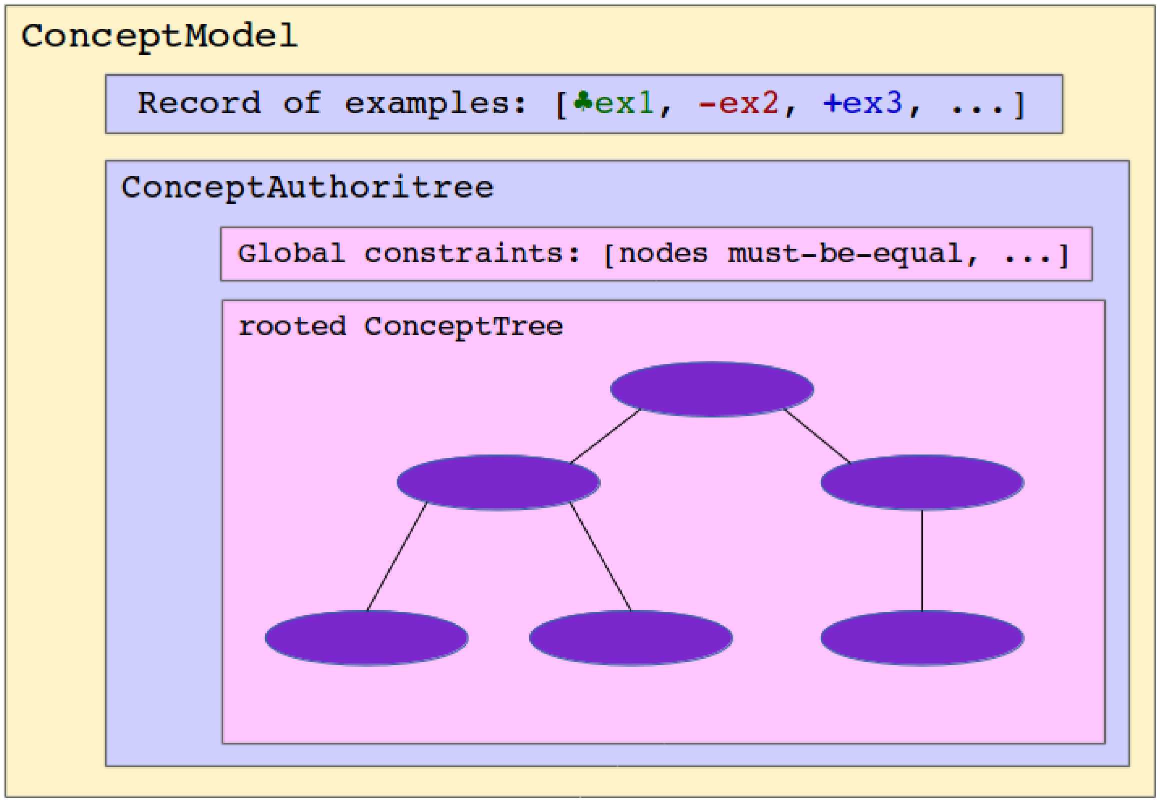

ConceptModel
Record of examples: [♣ex1, -ex2, +ex3, ...]
ConceptAuthoritree
Global constraints: [nodes must-be-equal, ...]
rooted ConceptTree

Figure 4-3: The `ConceptModel` stores the sequence of all examples given by the teacher, in addition to the `ConceptAuthoritree` which is the definitive, authoritative record of the concept itself. The `ConceptAuthoritree` maintains the global constraints that limit how different nodes of the recursively-defined `ConceptTree` interact, if at all.

examples as soon as they become reconcilable, if they weren't near misses at the time they were provided by the teacher.

4.3 A detailed look into STUDENT's inner workings

In this section, I will discuss the structure of the primary classes comprising STUDENT. First, I will motivate the discussion with a simple example.

4.3.1 A simple example

The atomic unit of a concept is the individual token, or *word*. Associated with each word is a bundle of threads representing the various word-senses associated with that word. Each node of the concept tree keeps track of which threads are and are not permissible in order for a candidate to match the concept.

Consider the following sequence of examples provided to STUDENT to teach the concept «*consume-food*»:

+Mary eats food.
+John eats food.

From these two examples, STUDENT concludes that «*consume-food*» must be defined roughly as:

Any person eats food.

To understand at a high level how STUDENT infers this model, first observe the primed threads associated with each of the words "mary", "john", "eat", and "food", illustrated in Figure 4-4.

Next, Figure 4-5 shows the basic tree structures of the two +positive example sentences. The *tree structures* only differ in one way: the **subject** node of the trees are different. STUDENT then attempts to reconcile that difference by first checking if the *nodes themselves* only differ in one way: in this case, they do, because the only difference between the nodes is the primed threads associated with each.

Figure 4-6 illustrates how STUDENT reconciles the two different **subject** nodes by comparing their primed threads. STUDENT finds that the semantic class "name" is common to both primed threads, and is in fact the *lowest* (most specific) common class between the two threads. As a result, STUDENT changes its model to indicate that *any* node whose primed thread contains class "name" will be permissible.

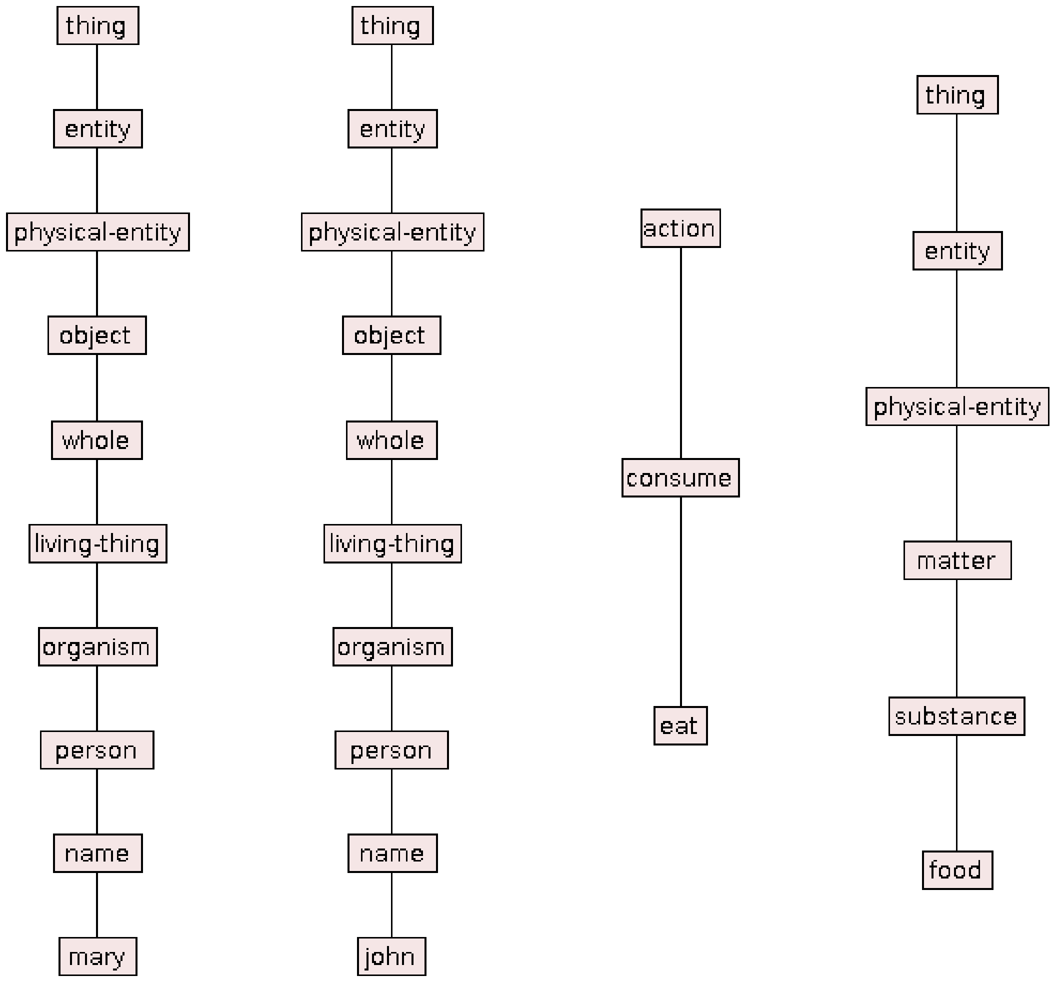


Figure 4-4: Each node in a concept pattern has a primed thread associated with it. Shown are the primed threads associated with the four tokens "mary", "john", "eat", and "food" respectively. Note how those for "mary" and "john" are very similar. STUDENT will soon take advantage of their similarity to reconcile them together as a single atom.

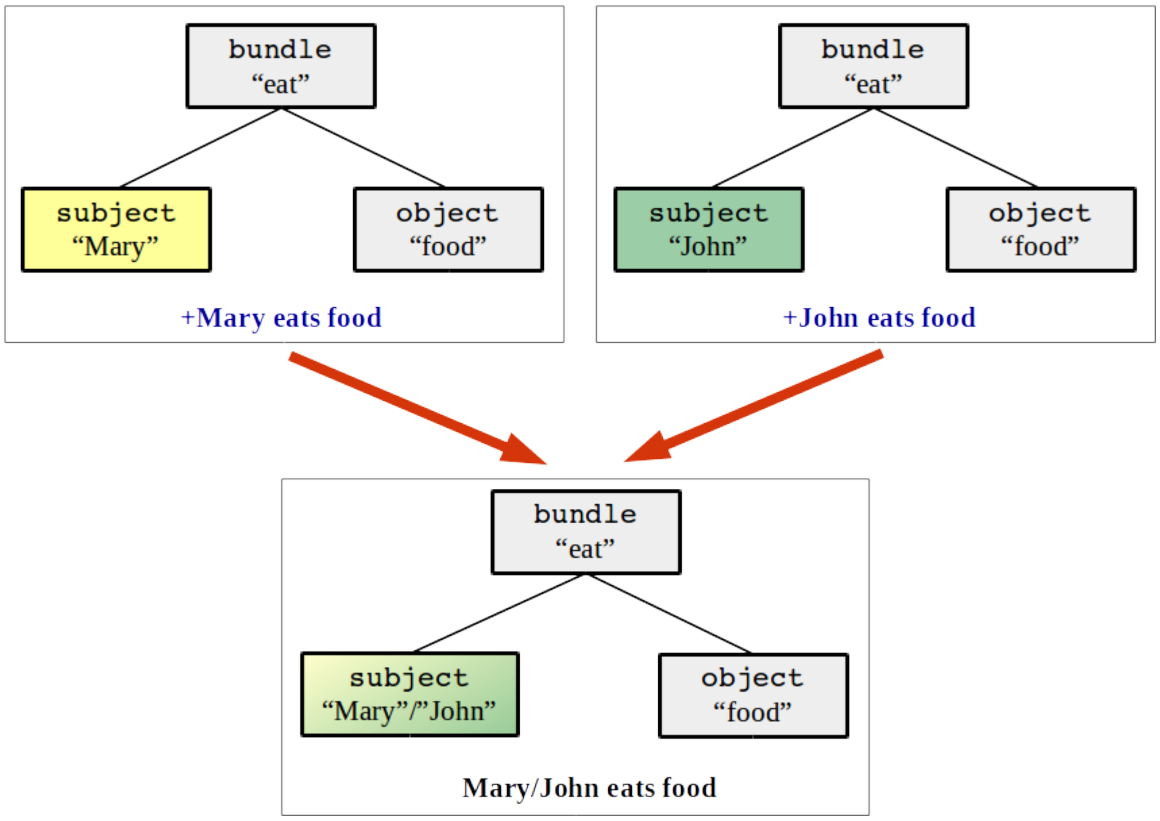


Figure 4-5: Two examples that differ in one key way can be reconciled. These two +positive examples differ only in the **subject** node. The two subject nodes themselves only differ in the token (that is, they differ in the primed threads associated with each), so STUDENT will now try to formalize a reconciliation between the two nodes to generalize its model of «*consume-food*».

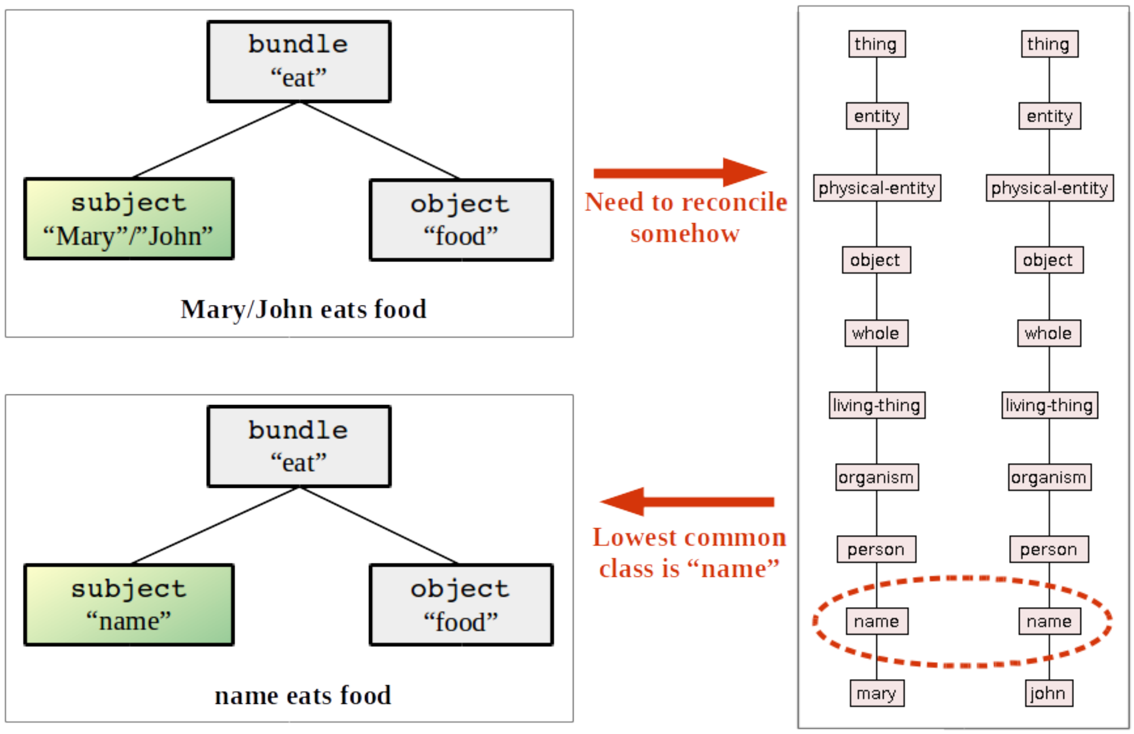


Figure 4-6: Two examples whose spools differ can be reconciled. STUDENT needs to find a way to reconcile these two differing nodes: it examines the primed threads associated with the two nodes, and finds that they share a common class "name". As a result, STUDENT generalizes the model at that node and now permits any candidate whose **subject** node is a token whose primed thread is "name" or any subclass thereof.

4.3.2 A `Spool` maintains a tree specifying permissible and forbidden words.

To formalize the concept of "keeping track of similar threads," I use a class called `Spool`, so called because it represents a collection of intertwined threads. Conceptually, a spool can be thought of as a tree formed by combining similar segments of different threads (in the style of Figure 2-1). However, a spool is a bit more complicated than that, because

- ▷ the spool must indicate which lowest-common-class specifications are permitted for the concept, and which are forbidden; and,
- ▷ STUDENT must be able to distinguish between +positive and –negative examples which impose generalization and specification constraints on the threads.

Spiritually, a spool is in some regards a fancy reification of a lattice-learning system as described by Klein (Klein, 2008) or Stickgold (Stickgold, 2011). However, the critical difference here is that the spool is only one small part of the data structure that encodes a concept model. Each concept will have many different spools associated with it, in addition to considerations for the structure of the concept itself.

To explore the properties of `Spool`s and expand on the above statements, consider first the concept model for «*consume-food*» resulting from a single seed example

♣Mary eats food.

The concept model now believes that the definition for «*consume-food*» is exactly

> *consume-food* $\Longrightarrow$ Mary eats food.

Given this model, would you expect "Mary eats matter." to be considered a manifestation of «*consume-food*»? No, of course not—*even though "matter" is listed as a class in the primed thread for "food"!* In other words, at this point in time, the model only accepts the most specific class possible for each of the threads.

The spool for the **object** node of this concept tree encodes exactly this information: it recognizes that a candidate whose **object**'s primed thread is at least as specific as "food" should be permitted; otherwise, it should be rejected, as visualized in Figure 4-7. One important conclusion from this is that, during example reconciliation, a ♣seed example is treated as a +positive example, because the seed gives initial information on what is *permissible* for a concept. This is a strict requirement in STUDENT's interaction:

The first (♣seed) example must demonstrate what is permissible.

To begin teaching a concept by giving a non-example would violate the Maxim of Quantity. After all, if someone asked you "What is a pickle?", you (probably) wouldn't respond by asserting

"A pickle is not an ostrich."

Spool updates become more complicated as the teacher supplies more examples. In general, STUDENT approaches permissibility conservatively, only permitting classes that have been directly or indirectly permitted; classes about which STUDENT has no information are assume to be forbidden. To determine if a thread class is permissible, the spool asks the following questions:

1. Is the class a terminal class? If so, then it was explicitly provided as an example by the teacher. Mark it as permissible if it was a +positive (or ♣seed) example, and mark it forbidden if it was a –negative example.
2. Otherwise, are any terminal descendant nodes of this class –negative terminals? If so, mark this node as forbidden. If not, mark this node as permissible.

Figure 4-8 describes a more complicated interaction. At top, we see the four spools corresponding to four given examples: in order, ♣mary, +dog, –rock, –robin.

1. STUDENT begins by initializing the model to mirror ♣mary.
2. Then, STUDENT reconciles the next example, +dog, with the model. It finds the common class "organism" and generalizes the model, assuming that *any* organism is permissible.
3. When reconciling the third example, –rock, nothing actually happens, because the common class is "object" which was already marked forbidden by default.
4. When reconciling the fourth example, –robin, STUDENT puts more import on recent restrictive constraints, re-forbidding "organism" and "animal" because they both have a –negative descendant terminal, "robin".

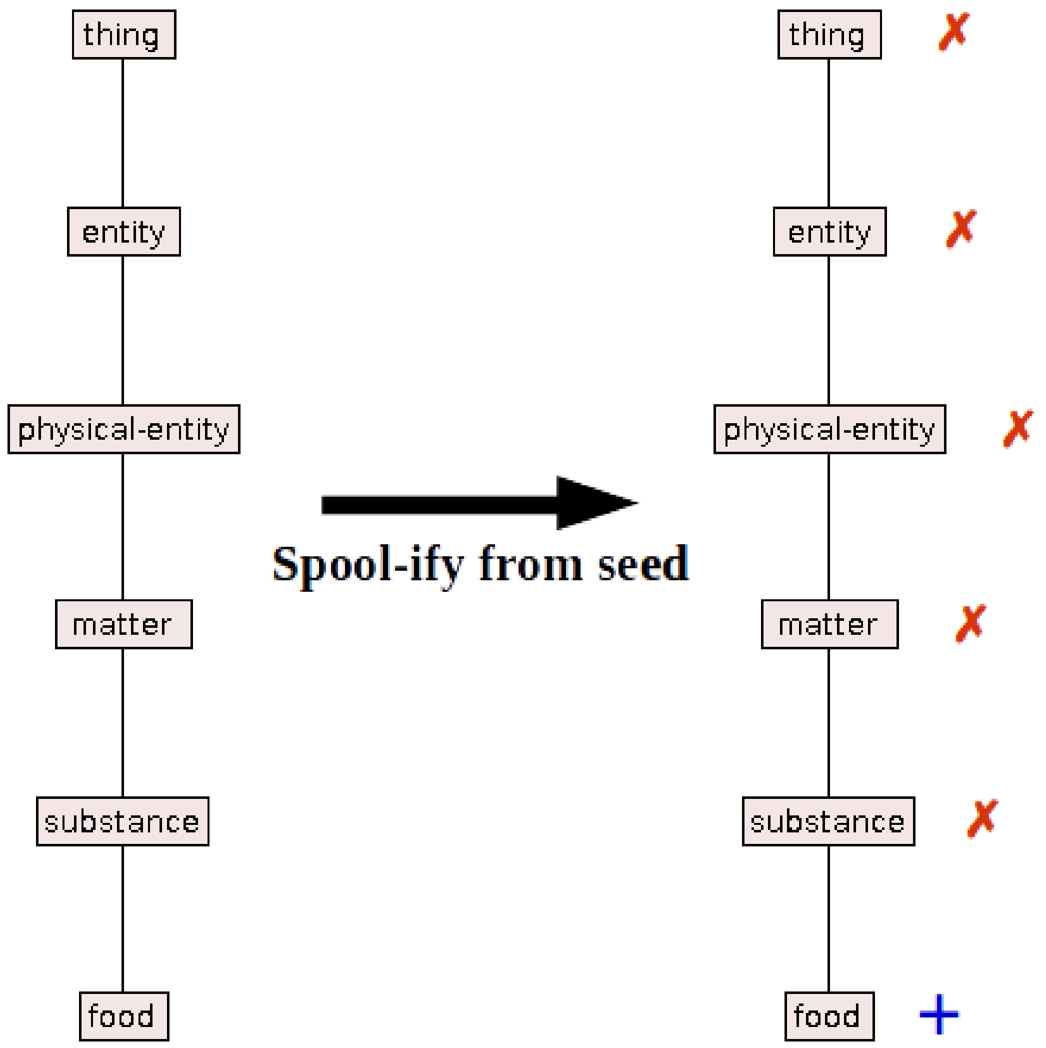


Figure 4-7: A spool formalizes permissible and impermissible minimal classes of a node's primed thread. On the left is the primed thread associated with "food" from the seed ♣Mary eats food., and on the right is the spool form of that thread: any candidate whose primed thread is at least as specific as "food" (indicated with a +, because it was supplied as an effective +positive example) is permitted; all others are rejected (indicated with a ✗).

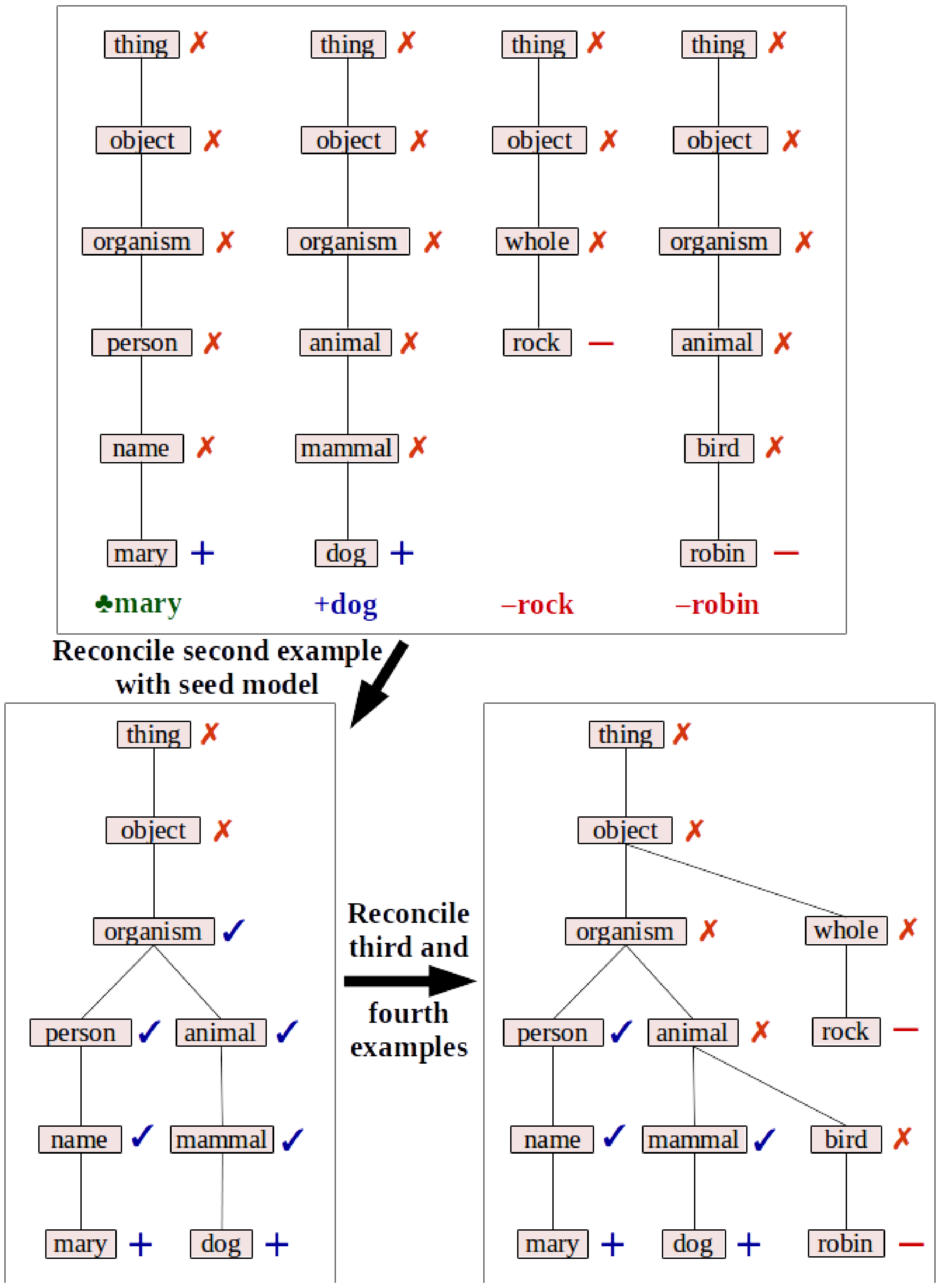


Figure 4-8: A spool can reconcile both positive and negative examples. At top, we see the four spools corresponding to four given examples. New examples are reconciled conservatively; sometimes, –negative examples override the permissibility of certain nodes in the spool.

4.3.3 A `ConceptTree` is an augmented `Entity`

Because the Genesis system internally represents Things as recursively-defined trees, so does STUDENT. A `ConceptTree` takes the important aspects of an `Entity` and retains them, and ignores fields that are largely irrelevant in the context of concept recognition (various annotations, identifiers, etc.).

Unlike `Entity`s and subclasses thereof, every `ConceptTree` instance has subject, object, and elements-like fields, though they may be `null` if not applicable. For example, an atomic entity like "happiness" has subject, object, and elements all `null`, whereas a relation like "harm" has subject and object non-null, with elements still `null` (because "harm" isn't a `Sequence`-type). I made this design choice because there is power in treating all concepts (Things) equally, allowing STUDENT to easily manipulate all nodes of a concept tree, and allowing general discussion of each atom of a concept.

Rather than storing the elements associated with a `Sequence`, a `ConceptTree` must be more general: it must also be able to indicate *forbidden* sequence elements, as well as *optional* sequence elements. For example, if «*physical-activity*» is defined as

> *physical-activity* $\Longrightarrow$ John throws Mary.

then it doesn't make sense to reject a candidate like "John throws Mary into the air.", as this is exactly the same but only with additional information! Hence, the concept model must know that some functions are optional—such as the function "into the air".

On the other hand, consider the following sequence of two examples defining «*physical-activity*»:

♣John throws Mary.
–John throws Mary under the bus.

The second example, –John throws Mary under the bus., is likely an idiom meaning that John betrays Mary; it is probably not implying that John literally tosses Mary underneath a vehicle. Hence, it is not a case of «*physical-activity*». With this example, STUDENT now knows to forbid a candidate that has the function "under the bus". This illustrates a situation where it is also important to maintain a record of forbidden elements.

In total, each `ConceptTree` node maintains a record of what types of elements are **required**, **forbidden**, or **optional**; I refer to these three types of constraints as the **existence qualifiers**, or just qualifiers for short. However, for ease of computation and generality, a `ConceptTree` doesn't actually associate full functions with each of the qualifiers: instead, it just specifies the *spool* of the rooted function node that is associated with the qualifier. In the above example, therefore, the three qualifier mappings are as follows:

```
required:  {object}
forbidden: {under}
optional:  everything except {object,under}
```

That is, an "object" is required; any prepositional phrase with preposition "under" (technically, any function with role "under") is forbidden; and anything that is not an "object" or "under" is optional.

The qualifier mappings has one important invariant: each role (e.g. "under") is associated with *exactly* one of the three qualifiers. This makes sense intuitively, as something cannot be (say) both forbidden and optional, but it has some repercussions as will soon be discovered.

Importantly, **the node separately maintains a table mapping each role to the structure of `ConceptTree` that must be satisfied**: these maps are stored in fields called `reqElePatterns` (for required functions) and `optElePatterns` (for optional functions). For consistency and possible future extension, there is also a corresponding `frbElePatterns`, but STUDENT actually just rejects any candidate that has a role matching a forbidden spool; the candidate's role function is not checked against a `ConceptTree`.

In the example above, the three fields look like:

```
reqElePatterns: {object: ConceptTree("object Mary")}
frbElePatterns: {under: (any ConceptTree)}
optElePatterns: {}
```

Note that `frbElePatterns` doesn't ascribe any particular structure to the "under" role, as STUDENT will automatically reject any candidate that expresses any "under" function whatsoever. Observe also that `optElePatterns` is empty: this is because all of the other roles are *implicitly* optional; STUDENT only assigns concept tree structures to explicitly-defined roles.

In general, when comparing a candidate to the concept model,

▷ For each role function the model requires, the candidate must have that function *and it must match the `ConceptTree` associated with that role*. That is, candidate

"John throws Mary."

would match the model described above, but candidate

"Mary throws a rock."

would not, because even though the role "object" is present, the `ConceptTree` associated with that function does not match the `ConceptTree` on record for "object".

▷ For each role function the model forbids, the candidate must not have any function with that role, regardless of the underlying `ConceptTree` structure for that node. For example, candidate

"John throws Mary under the stars."

would still be rejected, even though the `ConceptTree` associated with "under the stars" does not match the one associated with "under the bus".

▷ For each role function the model lists as optional, the candidate may or may not have a function with that role. However, if the candidate has a function with that role, *it must match the `ConceptTree` associated with that role*. For example, candidate

"John throws Mary into the pond."

would be permissible, because it introduces an optional role ("into") for which the model has no explicit model.

There are two drawbacks to this overall design choice:

1. It forces a role to be "either required or forbidden" or "either optional or forbidden," as any required or optional role may still be denied if it does not match the model

concept tree on record. It does not allow the model to differentiate between required and optional functions for a particular role.

2. It does not permit any flexibility for forbidden roles. Once a role is forbidden, it is impossible for *any* function with that role to ever be permitted. This isn't an accurate emulation of true concept learning, as there are often exceptions to forbidden statements.

That having been said, I believe this design choice offers a reasonable compromise between fidelity, generality, and speed.

To summarize, each `ConceptTree` has the following fields:

▷ `spool`: the `Spool` associated with this node

▷ `subject`: the subject of this relation/function (another recursively-defined `ConceptTree`)

▷ `object`: the object of this relation (another recursively-defined `ConceptTree`)

▷ `not` (not discussed): a flag indicating if there is a "not" modifier attached to this node. For example, in "does not harm", the node associated with "harm" has its `not` field indicating a negation.

▷ `existenceQualifiers`: The spiritual successor of "elements". Indicates what role functions (formerly, elements) are required, forbidden, and optional.

▷ `reqElePatterns`: Of the required roles, records the format of concept tree that must be satisfied.

▷ `frbElePatterns`: Of the forbidden roles, records the format of concept tree that must be satisfied. STUDENT doesn't currently use this field, but it's included to make it easier for a future version of STUDENT to take advantage of it.

▷ `optElePatterns`: Of the optional roles, records the format of concept tree that must be satisfied.

4.3.4 The `ConceptAuthoritree` affirms global constraints

The `ConceptAuthoritree` is an envelope class for the `ConceptTree`. Its primary purpose is to enforce global constraints among multiple nodes of the concept.

Suppose the following two examples are provided to describe «*self-harm*»:

♣Mary harms Mary.
+John harms John.

Recall, in the discussion of the student–teacher covenant (Section 3.2), that the teacher should not choose examples exhibiting certain characteristics unless such characteristics are relevant; furthermore, the student should assume that certain characteristics illustrated in an example are relevant to the concept at hand. Pursuant to these tenets, STUDENT assumes there is a reason that the provided example ♣Mary harms Mary. has the same (grammatical) subject and object. Upon seeing that those nodes are equal, STUDENT enforces a must-be-equals relation between them, as illustrated in Figure 4-9. Upon receiving the second example, +John harms John., STUDENT reconciles it as expected, generalizing both nodes to "name" while continuing to require that they be equal.

Similarly, imagine the opposite situation, where the teacher first provides a ♣seed example priming the model, then a –negative example enforcing additional constraints on the model. Consider the following sequence of four examples provided by the teacher to teach the concept «*harm-someone-else*»:

♣Mary harms John.
+Mary harms Sue.
+John harms Mary.
–Mary harms Mary.

STUDENT's thought process is explained below:

- ▷ ♣Mary harms John.: Concept model is initialized. STUDENT believes that «*harm-someone-else*» is exactly defined as "Mary harms John."
- ▷ +Mary harms Sue.: Concept model generalizes the grammatical object. Now, Mary may harm any person.
- ▷ +John harms Mary.: Concept model generalizes the grammatical subject. Now, any person may harm any person.
- ▷ –Mary harms Mary.: STUDENT is told that "Mary harms Mary." is forbidden, even though it appears to match the current model belief. STUDENT therefore assumes

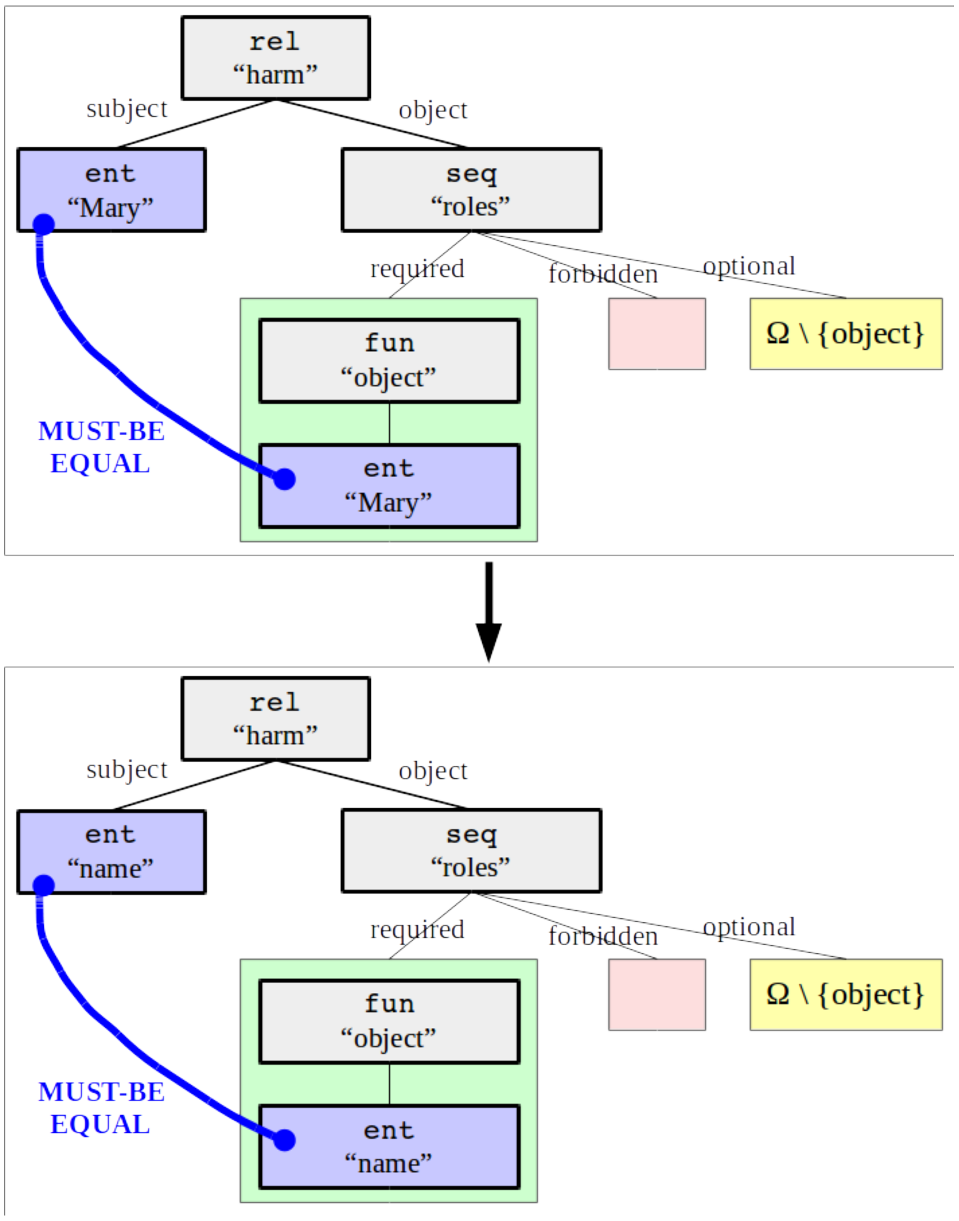


Figure 4-9: The `ConceptAuthoritree` enforces global constraints between concept nodes. At top, you see the full concept tree corresponding to ♣Mary harms Mary.; I have annotated each node with the type of `Entity` it emulates, though the true `ConceptTree` does not store such information explicitly. STUDENT asserts that, because the ♣seed example has two equal nodes (nodes with the same spool), the model should require the nodes to always be equal. Upon receiving the next example, +John harms John., STUDENT reconciles them as expected, maintaining the must-be-equal global constraint.
In this figure and in future figures, Ω refers to the full universe, and \ is the set-difference operator. For example, $\Omega \backslash \{x\}$ represents "everything except x".

that there must be relevance to the fact that the grammatical subject and object are identical. To reconcile the model with this new example, STUDENT adds a restriction to the model, forbidding any situation where the grammatical subject and object of "harm" are the same.

I have demonstrated two different situations where STUDENT properly heeds structural constraints alluded to by the teacher's well-chosen examples.

4.3.5 The `ConceptAuthoritree` oversees model transformations

The `ConceptAuthoritree` additionally oversees the incorporation of new examples into the model. In order to reconcile a new example with the current model, STUDENT must determine

1. if the new example is a near miss, and
2. if so, how to transform the model in a way to incorporate the new example.

It is not straightforward to determine if a new example is a near miss. Both the concept model and the example candidate are represented as complicated, recursively-defined trees; the trees must be compared in a way that highlights and quantifies their principle differences. If there are too many differences, STUDENT should conclude that the example is **not** a near miss. So, two questions must be addressed:

1. What exactly is a *near miss*? In particular, what types of maximal differences between trees should be permissible?
2. How are these differences to be found?

To answer both of these questions, I first experimented with some dynamic programming tree-difference algorithms. I quickly found that these algorithms grew too complicated as the definition of near miss became more nuanced (such as when multiple nodes can change at the same time when reconciling global constraints, as in Section 4.3.4).

Instead, I decided it was best to flip the problem on its head: instead of using a dynamic programming algorithm to find differences between trees, STUDENT should search through a concept-tree-space, applying all permissible "differences"—transformations—to the current model tree until the candidate is found. If the candidate is never found, STUDENT can conclude that the candidate is not a near miss.

This tree-space search is a brute-force search in spirit, but in style it is a branch and bound search. The search proceeds as follows:

1. Initialize the search agenda with the path containing just the current concept model's tree.
2. Pop the next best path from the agenda.
3. Extend the current path by applying each of the different transformation types to the most recent tree on the agenda, but only if the transformation cost so far isn't "too high." Put these new paths onto the agenda.
4. Repeat steps 2-4 until the agenda is empty (indicating the candidate is not a near miss) or the candidate tree is found (indicating the candidate is a near miss).

To improve efficiency, the search implicitly ignores loops and uses an extended set to avoid exploring already-explored trees.

I have defined eight different permissible transformations—ways to atomically transform a tree into a different tree—outlined in Figure 4-10. Associated with each transformation is a cost, indicating how different the tree is after applying the transformation. The cost of a sequence of transformations is the sum of the costs of each individual transformation; STUDENT designates a maximum-permissible transformation-path-cost of 1, suggesting that a near miss is "at most a cost of 1 away from the current model." In the style of branch and bound, the search always explores the path with the lowest transformation-path-cost-so-far. Figure 4-11 and Figure 4-12 give some examples of transformations, and Figure 4-13 illustrates a situation where the search is enacted but concludes that the candidate is *not* a near miss.

- ▷ `change-spool` (0.98): Changes the spool of a particular node to a different spool.
- ▷ `change-spools` (0.99): Changes the spools of two (or more) must-be-equals nodes to a different set of equal spools.
- ▷ `toggle-not` (1): Flips the `not` flag on a particular node.
- ▷ `swap-subject-object` (1): Swaps the *grammatical* subject and object of a verb (**not** the `Relation`'s subject and object).
- ▷ `add-element` (1): Adds a new element to a sequence-like node by changing it from **forbidden** to **optional**.
- ▷ `remove-element` (1): Removes an element from a sequence-like node by changing it from **required** to **optional**.
- ▷ `change-element` (1): Changes an element to a different element in a sequence-like node.
- ▷ `expand-sequence` (0.2): Turns a non-sequence-like concept node into a sequence-like node, inserting the former `ConceptTree` associated with the node as the sole required element of the newly-created node. This is theoretically useful when manipulating concepts that introduce (say) conjunctions in new examples, but hasn't seen much use in practice. See Figure 4-11 for an example.

Figure 4-10: A *near miss* is a candidate that can be created by transforming the current model's `ConceptTree` only a certain number of times. The eight ways to transform a `ConceptTree`, and their associated costs, are shown above. If the `ConceptTree` can be transformed into the candidate via a series of transformations whose total cost is at most 1, the candidate is considered a near miss. Otherwise, the candidate is not a near miss.

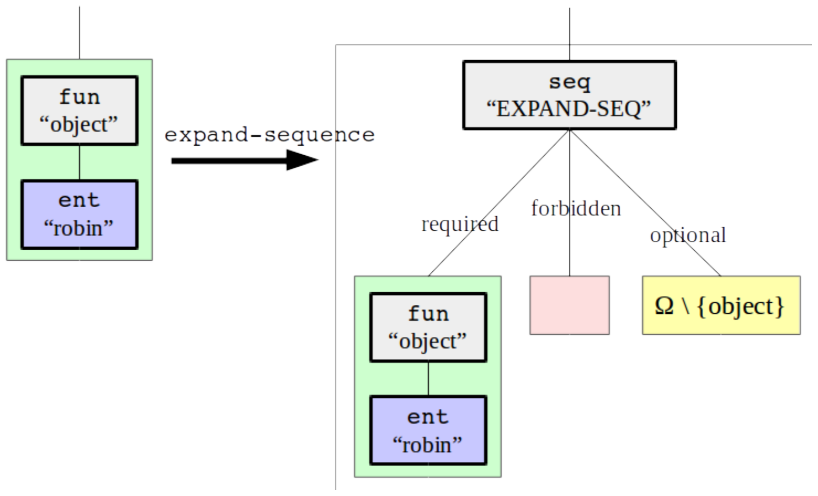


Figure 4-11: The `expand-sequence` transformation allows concept generalization for a particular concept node. Once a node has been expanded to a sequence, it may be easier to transform or match it against other similar sequences.

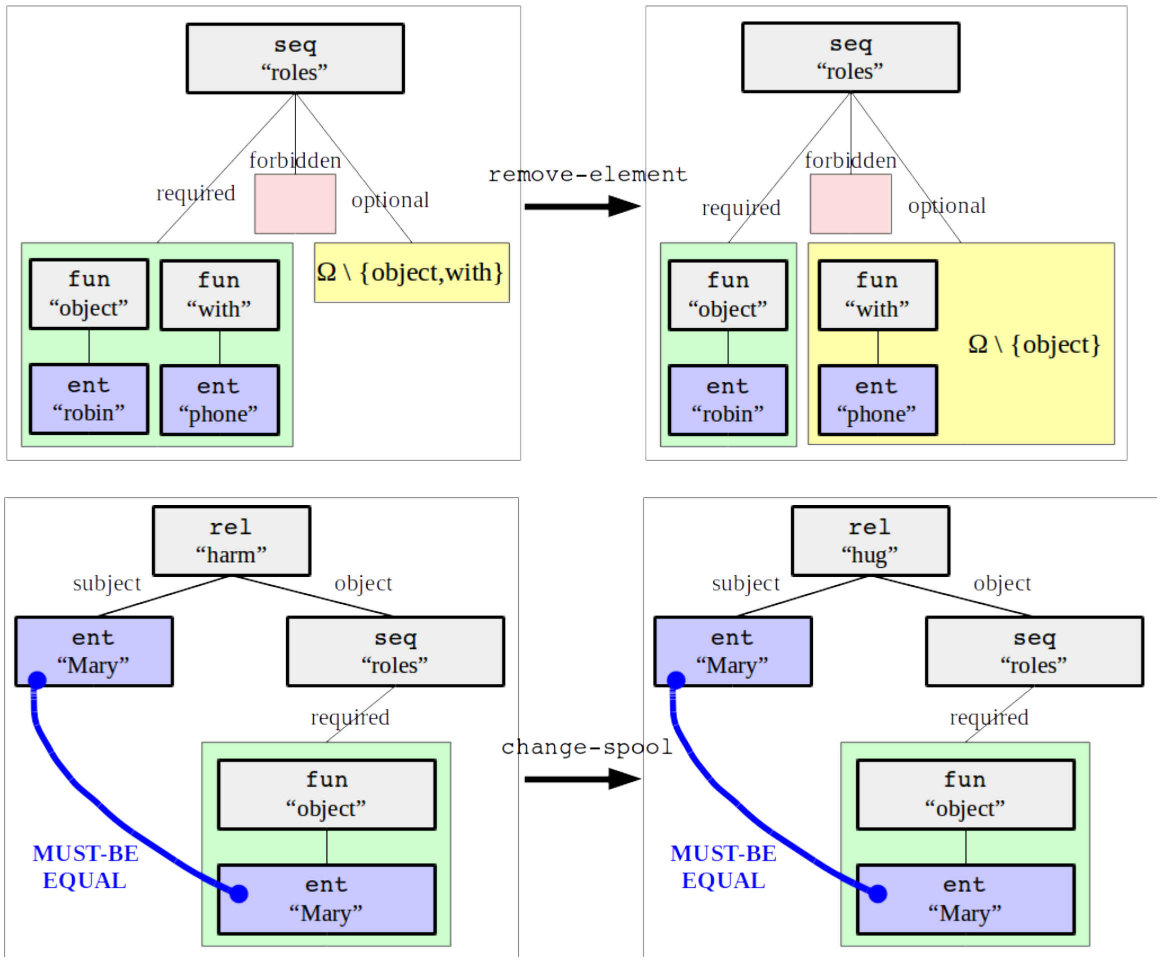


Figure 4-12: Different types of transformations allow different types of near misses to be reconciled. On top, the `remove-element` transformation is used to remove a required "with" function; the "with" function is now optional. (The *optional* box indicates that everything except "object" functions are optional; furthermore, any present "with" function must match the concept pattern given.) On bottom (for visual clarity, **optional** and **forbidden** fields are not shown), the `change-spool` transformation allows the verb "harm" to change into "hug".

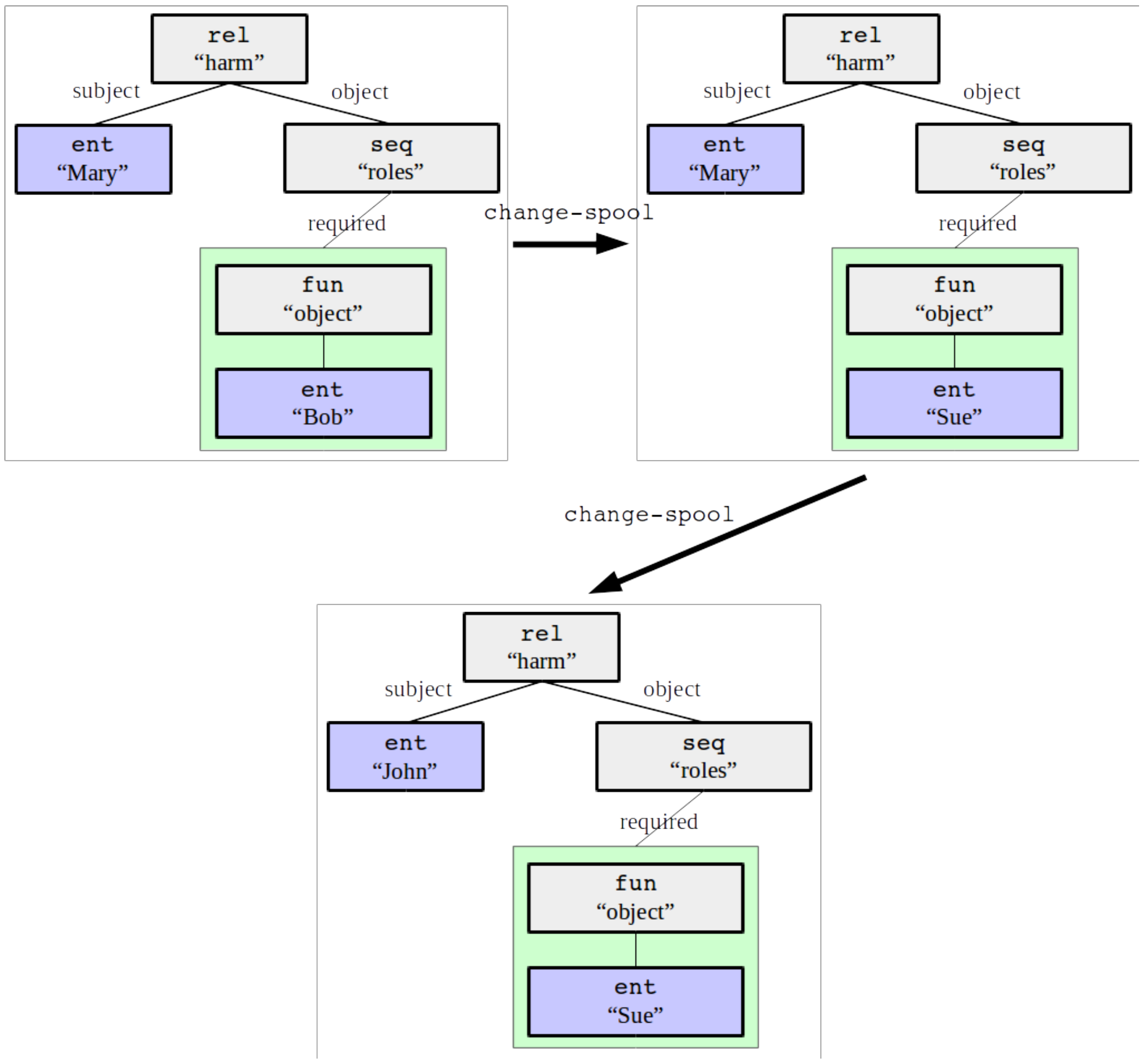


Figure 4-13: Sometimes, the search finds that the shortest transformation path is too costly, and concludes that the candidate is not a near miss. In this example, trying to reconcile "Mary harms Bob." with "John harms Sue." yields disastrous results. (For visual clarity, in the trees shown above, **optional** and **forbidden** fields have been hidden.) The only transformation path between these two trees is a sequence of *two* `change-spool` transformation paths, resulting in a total cost of 2. STUDENT concludes that "John harms Sue." is not a near miss.

Chapter 5

Contributions

Successfully teaching a system how to learn concepts by example is a large step towards understanding how to make human intelligence systems that can understand stories, because understanding new and emerging concepts is a large part of story understanding. Minsky briefly entertained the notion of the baby-machine, a system that could simulate a newborn baby learning about the world from physical stimuli and context clues without requiring manually-inputted common sense and idea definitions. Teaching a system to learn and reason on its own, by example, is a step in the direction of the spirit of this baby-machine.

With the arrival of STUDENT, Genesis is no longer reliant on hard-coded, manually-defined definitions of concepts; instead, a cognizant teacher can supply several examples of a concept to give Genesis an ostensive definition of the concept. Furthermore, creating a system that learns concepts by example opens up the possibility for "creative misunderstanding," where a machine learns a different definition of a concept from what the human teacher originally intended, but it ends up being a *better* definition of the concept. This is an important capability because it no longer puts the onus on the human to perfectly define everything—instead, the system can make decisions for itself.

In a way, STUDENT's example-oriented paradigm means that the user has to put in more work than before to teach a concept: providing several *good* examples of a concept may be more tedious than writing down a direct definition. However, the trade-off is that the burden is no longer on the user to design a perfect and exact definition of the concept. This is a powerful compromise, as it is much easier for two people to agree on some examples of a concept than to agree on the exact definition of the concept. With STUDENT's effort, the guesswork is taken out of the hands of the human, and is instead provided as organic learning stimuli to the human intelligence system.

Bibliography

AbstruseGoose (2012). Arithmetic for beginners. *Abstruse Goose*. Accessed: 2018-01-02.

Grice, H. P. (1975). Logic and conversation. In Cole, P. and Morgan, J. L., editors, *Syntax and Semantics: Vol. 3: Speech Acts*, pages 41–58. Academic Press, New York.

Katz, B. (1993). Start. *START Natural Language Question Answering System*.

Klein, M. T. (2008). Understanding english with lattice-learning. Master's thesis, Massachusetts Institute of Technology.

Krakauer, C. E. (2012). Story retrieval and comparison using concept patterns. Master's thesis, Massachusetts Institute of Technology.

Miller, G. A. (1995). Wordnet: A lexical database for english. *COMMUNICATIONS OF THE ACM*, 38:39–41.

Minsky, M. (2006). *The emotion machine : commonsense thinking, artificial intelligence, and the future of the human mind*. Simon & Schuster, New York.

Stickgold, E. (2011). Word sense disambiguation through lattice learning. Master's thesis, Massachusetts Institute of Technology.

Vanlehn, K. A. (1983). *Felicity Conditions for Human Skill Acquisition: Validating an Ai-based Theory*. PhD thesis, Cambridge, MA, USA.

Winston, P. H. (1970). *Learning Structural Descriptions from Examples*. PhD thesis, Massachusetts Institute of Technology.