Genesis’s implementation substrate

Patrick H. Winston
7 October 2014

Sources and deprecation of System.out.println

You can run the code examples given here by way of demonstration class in the tutorials package in the Genesis system.

Note that we do not use ordinary print statements in genesis. The reason is that our fancy print statements print on the Eclipse console with a clickable pointer to the place in the code where the fancy print statement lies. Hence:

Genesis is implemented on a substrate of four Java classes

• The Entity class, so called so as not to conflict with Java’s Object class. A entity instance has no internal structure, other than a bundle of threads (unrelated to Java’s Thread class; alas, a name clash we have decided to live with), which specify class membership. Entities are used, for example, to represent particular physical objects.

• The Function class, so called so as to suggest that instances are derived from an object.

• The Relation class. A relation indicates how one object is connected to another object.

• The Sequence class. A sequence contains any number of elements.

From the Java perspective, functions are entities and relations are functions. Sequences are also entities. Thus, because entities have threads, so do functions, relations, and sequences.

The Entity class

The most fundamental class is the Entity class. Every instance has a unique name and bundle of threads.

To create a new entity, identified as both a ball and a baseball, you execute the following:

Entity x = new Entity("ball");
x.addType("baseball");

Then, you can inspect the result via a print statement:

Mark.say(x.toXML);
--> 
<entity>
  <name>baseball-0</name>
  <bundle><thread>entity ball baseball</thread></bundle>
</entity>

Alternatively, if you just print the entity, you get less, but more readable information:

(ent baseball-0)
The first thread of possibly many threads in the thread bundle is considered the primed thread. Whenever you add a type to a entity, using the addType method, that new type goes to the end of the primed thread.

The isA predicate tests for class membership by checking all the threads. Hence \( x\text{.isa}("ball") \) returns true, but \( x\text{.isa}("person") \) returns false.

The Function class

The Function class is a direct subclass of the Entity class, used, for example, to represent Jackendoff’s places and path elements. The Function class provides getSubject and setSubject accessors. The subject is to be filled with an Entity instance or an instance of an Entity subclass. The following example shows how to construct a function that represents the top of a table:

```java
Entity e = new Entity("table");
Function f = new Function("top", e);
Then:
Mark.say(f);
-->
(fun top (ent table-1))
Mark.say(f.getSubject());
-->
(ent table-1)
```

The Relation class

The Relation class is a direct subclass of the Function class. The Relation class simply adds an object slot to the Function class, with getObject and setObject accessors.

```java
Entity d = new Entity("door");
Entity w = new Entity("window");
Relation r = new Relation("between", d, w);
Mark.say(r);
-->
(rel between (ent door-4) (ent window-5))
Mark.say(r.getSubject());
-->
(ent door-4)
Mark.say(r.getObject());
-->
(ent window-5)
```

The Sequence class

The Sequence class is a direct subclass of the Entity class. Sequences appear, for example, when Genesis produces role frames from English. Given *John killed Peter with a knife*, for example, Genesis produces a sequence containing two roles, one for the object, Peter, and one for the instrument, the knife.

```java
Sequence roles = new Sequence("roles");
roles.addElement(new Function("object", new Entity("Peter")));
roles.addElement(new Function("with", new Entity("knife")));
Relation k = new Relation("kill", new Entity("John")), roles);
Mark.say(roles);
-->
(seq roles (fun object (ent Peter-8)) (fun with (ent knife-10)))
Mark.say(k);
-->
Convenience constructors build role frames

It can be tedious and error prone to construct role frames, especially when the object is filled with a path description, which itself is complex. Fortunately there are convenience constructors that help. The following, for example, shows how to create a role frame and how to add an additional role to an existing role frame:

```java
Relation rf = Constructors.makeRoleFrame(John, "kill");
Mark.say(rf);
-->
(rel kill (ent John-14) (seq roles))
rf = Constructors.makeRoleFrame(John, "kill", Peter);
Mark.say(rf);
-->
(rel kill (ent John-14) (seq roles (fun object (ent Peter-15))))
rf = Constructors.addRole(
    Constructors.makeRoleFrame(John, "kill", Peter), "with", knife);
Mark.say(rf);
-->
(rel kill (ent John-14) (seq roles (fun object (ent Peter-15)) (fun with (ent knife-17))))
```

The following shows how to create a role frame involving a path object:

```java
Entity tree = new Entity("tree");
// Create a place relative to entity
Function place = JFactory.createPlace("top", tree);
// Create a path element using place
Function pathElement = JFactory.createPathElement("from", place);
// Create a path
Sequence path = JFactory.createPath();
// Add path element
path.addElement(pathElement);
// Use path to create a trajectory
Entity trajectory = JFactory.createTrajectory(bird, "fly", path);
// Create another path element
Function origin = 
    JFactory.createPathElement("to", JFactory.createPlace("at", new Entity("rock"));
// Add it to the path
JFactory.addPathElement(path, origin);
// Have a look
Mark.say("Amended trajectory role frame: " + trajectory);
-->
Amended trajectory role frame: (rel fly (ent bird-16)
    (seq roles (fun object (seq path (fun
from (fun top (ent tree-44))
    (fun to (fun at (ent rock-71)))))))))
```

Genesis’s translator produces innerese frames from Genesis English

Having slogged through all the explanation of how to work with entities, functions, relations, and sequences, you are ready to learn that you generally do not have to ever construct such objects. Genesis does it for you
by translating from English:
String sentence = "John marries Mary because John loves money";
Translator translator = Translator.getTranslator();
Entity entity = translator.translate(sentence);
Mark.say(entity);
-->  
(seq semantic-interpretation
   (rel cause (seq conjunction (rel love (ent john-93)
      (seq roles (fun object (ent money-103)))))
   (rel marry (ent john-93) (seq roles (fun object (ent mary-100))))))

Genesis also features an English generator. The following thus produces the same English that is provided:
Generator generator = Generator.getGenerator();
Mark.say(generator.generate(translator.translate(sentence).getElements().get(0)));
-->  
John marries Mary because John loves money.

The Bonawitz view

The printed form of entities, functions, relations and sequences is far from transparent. Accordingly, when such instances are presented in a GUI they appear in a form conceived by Keith Bonawitz. The following illustrates:

Note the color coding: entities in gray, functions in blue, relations in red, sequences in black.
Note also that when you role over various parts of the diagram, you see the various threads Genesis associates with each entity.
Figure 1