Enhancing the IOA Code Generator's Abstract Data Types

A Report of Atish Nigam's 2001 Summer Work

This past summer I worked on a number of different aspects of the IOA Code Generator. While the initial part of the summer served as an introduction to the code generator, I was able to assist in producing more abstract data types and I was also able to help the group in other ways as well. My summer projects were divided into three main parts and it is these three parts that I will expand on and discuss in this report.

The Stack Abstract Data Type

The first project that I worked on was adding the Stack ADT to the reserve of ADT's that were already contained in the Code Generator. An ADT is an abstract data type that is used by the code generator as a template for converting IOA code into useable java code. The code generator works by parsing through an IOA program and identifying some of the features (such as ADT's). Then for each of these features, the code generator translates the code from the IOA abstract syntax tree to the Java abstract syntax tree. The ADT repository is very important as it serves as the dictionary for translating between the two abstract syntax trees.

Stack was implemented using the Vector data type where elements are added to the top of the stack and are removed from the top of the stack as well (where the first element in the vector is the top of the stack). To conform to this definition a few design choices had to be made in the ioa.runtime.adt.StackSort.

The java class java.util.Stack was relied upon immensely in the design of the Stack ADT runtime class, as it had many of the common methods that were desired such as pop and push. One design choice that was made in this process was choosing how to count the number of occurrences of an element in a given stack. To do this a clone of the stack was used and the stack was made into an Enumeration. Then the method ran through the enumeration comparing each element in the stack to the desired element and maintained a running count of how many occurrences of each element were found. A similar method was done to test if two stacks were equal. In this case if both stacks' lengths were equal, an enumeration was produced from each stack. The elements of each enumeration were then checked against each other to ensure that each of these were equal as well.

After writing the runtime class, a separate ioa.codegen.impl.java.StackSort class was written to install the Stack ADT into the code generator. After further work on other aspects of the code generator this summer, the installer class was moved thus, this implementation class has gone through significant changes. The new StackSort installer can be found at ioa.registry.java.StackSort and works with the new installer.

A test suite (ioa.test.junit.runtime.adt.StackSortTest) was added to the code generator to test to make sure that the current implementation works correctly. Also this test suite ensures that if further changes were made to the Stack ADT, these changes would not disturb the functionality of the total ADT. One example of things that this file tests for is ensuring that after popping from the stack, the order and hierarchy is maintained in the resulting stack. The last addition was a test IOA file that was written, Stack01.ioa, to ensure that all the files worked and did what the code generator desired. This file is

essentially a very simple IOA automaton that has IOA representations of all the features that the Stack ADT supports. The code generator runs through this file and translates all of this IOA code to java. The result of this translation is used to see if the code generator can correctly identify Stack representations in IOA code.

While the Stack ADT was very simple to implement, the writing and execution of this class was very important to my learning of IOA and the IOA Toolkit. From my work on this I was able to branch out later in the summer and work on the other aspects of the toolkit.

The Priority Queue ADT

The next step after working on the Stack ADT was to install the Priority Queue ADT. I modeled this ADT after the description of the binary heap in the book Introduction to Algorithms by Cormen, Leiserson and Rivest. The Priority Queue is a representation of a queue in tree format. The elements that are highest priority (or lowest if it is a minimum binary heap) are put at the top of the queue and then the rest of the elements are added to the binary tree in decreasing order (or increasing for the min heap). Thus at any node in the tree, all elements that are at lower nodes are of lower priority. In this description, the Priority Queue was implemented with the highest priority element at the top of the queue and then the next lowest elements at lower branches in order of highest priority, this is known as a maximum binary heap data structure. After further work this summer a minimum heap option was also added. This option needs a little more work to ensure seamless addition to the whole toolkit, as it is currently a simple hack at the moment.

The Priority Queue proved to be much more of a challenge than the Stack ADT, as there are no Java API's that implement a binary heap and there were no ADT's previously written that implemented a heap either. The queue was stored as a java vector and the elements at each position on the vector corresponded to positions in the heap (position 0 was the head of the queue, position 2 was the right child of the head of the queue, and so on).

The first design choice that was made was regarding counting the number of occurrences of a specific Comparable ADT in the priority queue. To do this, count and helper methods were implemented. The count method calls the countHelp method to find the first occurrence of the desired ADT. The countHelp function then increments a counter by one and finds the next occurrence of the desired ADT. This countHelp method parses through the whole queue and then returns the number of times the desired ADT was found. The next design choice that was made was regarding the method that tests equality. This method was done in a similar way to the Stack ADT where enumerations of each Queue were made and then each enumeration was run through to ensure that each queue had the same number of elements and that each queue had identical elements. Before each queue was made into an enumeration, however, each vector was sorted to list the elements in order from greatest to least. This was done because it is possible for two equal trees to have elements in different positions. Thus, to ensure that all trees are equal each tree was sorted and treated like a vector.

Another valuable tool that was added was the checkHeapProperty method and this method checks to ensure that each priority queue is a correctly formed binary heap. This

is used by methods to ensure that when an element is added or removed from a queue, the resulting queue is well formed. If a queue is not well formed, the heapify method is used to convert the queue into a well-formed binary heap. Heapify essentially takes a tree and adjusts it to obey the heap property if it doesn't already.

The next important method of the Priority Queue is the add method. This method takes the given element that is to be added to the queue and adds it to the last open slot of the queue. Then this method checks to see if the element is larger than its parent. If it is larger than its parent, it swaps places with its parent. After this, it checks to see if it is larger than its new parent if this is true then these elements switch again. This keeps happening until the newly added element cannot progress to a higher position in the queue. Next, the checkHeap method is run to test if the queue has any bugs and the rest of the queue is adjusted to ensure that the queue still obeys the heap property. The tail method was the last method that was implemented. This method simply returns the last element of the queue, removes this from the queue and adjusts the heap structure.

Some other private utility methods that were implemented in this class were leftChild and rightChild, which return the given elements that are children of an element that the user specifies. Also along the same line are hasLeftChild and hasRightChild, which return booleans based upon if an element, has a child or not.

Finally as in the Stack ADT a given set of subsequent classes were written. The test suite was written and stored in ioa.test.junit.runtime.adt.PQSort. The installer class was written and stored in ioa.codegen.impl.java.PQSort and has now been moved to ioa.registry.adt.PQSort to conform to the new installer classes that were implemented. Finally, an ioa file was written to test that the installer and ADT's are incorporated well into the code generator. These files are stored in PQ01.ioa and PQ02.ioa.

The Priority Queue implementation is now very streamlined and concise however, it did take me a good deal of time writing and then re-writing code to get it to this stage. The current implementation is so far the best but to get to this implementation, there were many different versions and situations that I tried. Through this process I was able to not only learn about the different implementations and possible scenarios with a Priority Queue, but was able to learn a lot about java and different java types that I used to further simplify my code.

The ADT Locator and ClassLoader

The biggest and most time consuming project that I undertook this summer was writing an ADT Locator and Loader function, which automatically installs the ADTs that the user desires to be a part of the code generator. This was important because as the code generator expands and becomes more popular it is thought that users will write their own ADTs and add them to their own directories. These classes find all of these ADTs and then install them into the code generator so that the code generator can locate implementations of these ADTs in each of the IOA files it looks at. These files find their configurations from a new dotfile that was added to the toolkit, called .ioarc. This file details the location of all the ADTs that are to be loaded into the code generator. There are a number of possible options in the .ioarc file that are detailed here:

- *ioa.registry.locationOfPackages:* This option is used so that the user can specify where ADT class packages are possibly located, so that in the installation process, the code generator knows where to look. The default for this is IOA_Toolkit/Code/classes/

- *ioa.registry.listOfPackages*: This option is used by the code generator to detail which full packages and jar files should be loaded into the code generator. The default for this is ioa.registry.java. This is useful if the user has made a package of ADT classes that are to be loaded and he has not added these to the ioa.registry.java package (it is expected that as users add their own ADTs they will not add them to the given ioa.registry.java package but will rather add them to another package).

- *ioa.registry.classesToExclude*: These are the undesired classes that may be found in some of the packages that were loaded above. This is so that if a user doesn't want a specific class to be loaded but wants all the other classes in that package to be loaded, he can ask the code generator to load that package and ignore loading a given set of classes.

- *ioa.registry.classesToInclude*: These classes are ones that the user wishes to load which were not included in their above list of packages to load. This is so that if the user writes a single specific ADT class this can be added to the code generator.

- *ioa.registry.registrableInterface*: This is the registrable class. Each ADT that is implemented and is registrable should implement the registrable interface. This option allows the user to expressly list where the registrable class is located so that the ADT Locator class can ensure that each ADT that is desired to be loaded implements this class. The default for this is ioa.registry.Registrable.

If a user wants to simply use the IOA defaults, a catch is included in the ADT Locator class to load the default classes if a user does not have an .ioarc file. Additionally the user has the option to add two more parameters at the command line prompt *ioa.registry.moreClassesToExclude* and *ioa.registry.moreClassesToInclude*. Both of these simply add to the *ioa.registry.classesToExclude* option and the i*oa.registry.classesToInclude* category and are useful if the user simply wants to temporarily add or subtract ADT classes.

This .ioarc file is used by two subsequent classes to find and load the correct ADTs. The first class is the ClassLocator class (ioa.util.ClassLoader), this file searches for specified directories and jar files looking for classes that implement a specific interface. It takes as inputs a collection of packages and classes. Some of these detail classes that are to be included and loaded, while other collections list a few classes from specific packages that are not to be loaded into the Toolkit. The class loader was specifically designed to be abstract to ensure that it could be used for other functions as well.

As it pertains to locating ADT's the ClassLocator first goes through the list of packages to be loaded (from *ioa.registry.listOfPackages*) and finds where each are located on a users computer (from *ioa.registry.locationOfPackages*) using the jarAndPackageLocator

method. This method takes a file name and an iteration of possible locations and returns the location of the file. It uses several helper methods such as jarClasses and packageClasses which both take a path to a jar file or to a package and return a list of classes that are contained in these files. This creates a set of all of the classes in these packages.

The next step is to remove from this set all of the classes that the user wishes to exclude from being loaded with the removeClass method. After this is done the ClassLocator adds to this set all of the classes that the user has specified to add (from *ioa.registry.classesToInclude*), making sure that these classes are not already a part of the set. The method that does this is the addClasses method.

Finally the ClassLocator runs getImplementorsOf, which goes through each of the classes and checks to see that they implement the registrable interface (from *ioa.registry.registrableInterface*). If one of the classes in the set doesn't implement the registrable interface then this class is removed from the set. After this is done the ClassLocator returns the total list of classes that the user wishes to be installed and that implement the registrable interface.

The next step in this process is of course loading these classes. This is where the ioa.registry.ADTLoader class comes into use. The first thing this class does is to read the users .ioarc file to find what all the relevant options are. This class then calls ClassLoader with the .ioarc options and gets a set of all the ADTs that are to be implemented. Finally, it takes this set and runs the install procedure on each of the classes that are detailed in the set and ensures that these are in the necessary registries.

This portion of the summer took by far the longest as it contained a lot of writing and re-writing of code. The original plan that was designed for this was to have an ADTLoader and an ADTLocator class with a .adtrc file. After manipulating this for a while and realizing that other IOA coders could benefit from a universal dotfile in the IOA Toolkit, it was changed around and eventually came to be universal for IOA. Another thing that was very important to me about this portion of the summer was that I was able to really learn a lot about Java programming and coding style. Both Michael Tsai and Josh Tauber were helpful in making sure that I was coding correctly and following all the coding conventions that were desired.

Other Smaller Projects of the summer

Along the way this summer there were a few small projects that I picked up as well and spent some time working on. The first was making each of the ADTs that could possibly be loaded implement a function called ioa.registry.Registrable. This class is ensures that each of the classes that implement this has an install method inside of it. It is used as was described above to ensure that all the classes that were attempted to have loaded were loaded correctly.

A second thing that I did was along the same line. I expanded ADTs that were already written such as Integer, Real, Natural and Character, to have compareTo functionality. The coding changes I made was to ensure that each of these ADTs implemented ioa.runtime.adt.ComparableADT and that subsequently each of the ADTs had a

compareTo method inside it which simply returned a value based on what one data type was in relation to another data type. For example, if one IntSort is greater than another IntSort the compareTo function for IntSort would return a 1, and a –1 if it was less than and a 0 if the two were equal. This functionality was extremely important in writing the PQSort and other more recent ADTs, as most of these relied on being able to find relationships between two data points.

Another thing that I did this summer was to run benchmark tests on java's BigInteger and BigDecimal classes. This was done to see if switching or adding this functionality to the code generator would be a viable solution so that a user could use very large numbers. After running a few benchmark tests it was found that BigInteger and BigDecimal ran at ¼ the speed of java's Integer datatype for simple arithmetic such as addition and subtraction. After finding the results of these tests, it was determined that these data types would not be supported as changing the code generator would take too long and would create added headache. This is to be done later in the progress of the code generator.

Overall the summer was very enjoyable and I had a great time working with the IOA group. I look forward to continuing my work on the Toolkit this coming semester and taking a larger role in getting the project to the next levels.