Abstract Data Types for IOA Code Generation

Michael J. Tsai*
mailto:mjt@theory.lcs.mit.edu

August 16, 2001

*This work was supported in part by National Science Foundation contracts ACI-9876931 and CCR-9804665, Air Force of Scientific Research contract F49620-97-1-0337, and the Defense Advanced Research Agency contract F19628-95-C-0118.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Code Generation Overview

When the IOA[2] code generator[6] converts an IOA program into runnable Java code, it first parses the program, which has already been translated from IOA into the intermediate language (IL). The result of this parse is an abstract syntax tree with nodes for each element of the IOA program (e.g. transitions, state variables, and operators). The source syntax tree consists of objects from the `ioa.codegen.source.java` package.

The next step is to translate the source syntax tree, node by node, into the target syntax tree. The target tree consists of objects from the `ioa.codegen.target.java` package and has nodes that correspond to Java language elements (e.g. classes, variables, and methods). In the final step, the code generator walks the target syntax tree asking each node to emit itself as a string. The result of emitting the whole target tree is a Java program that can be compiled and run (with the help of classes in the `ioa.runtime.adt` and `ioa.runtime.io` packages).

The relation between the different parts of the code generator is summarized by the package diagram in Figure 1.

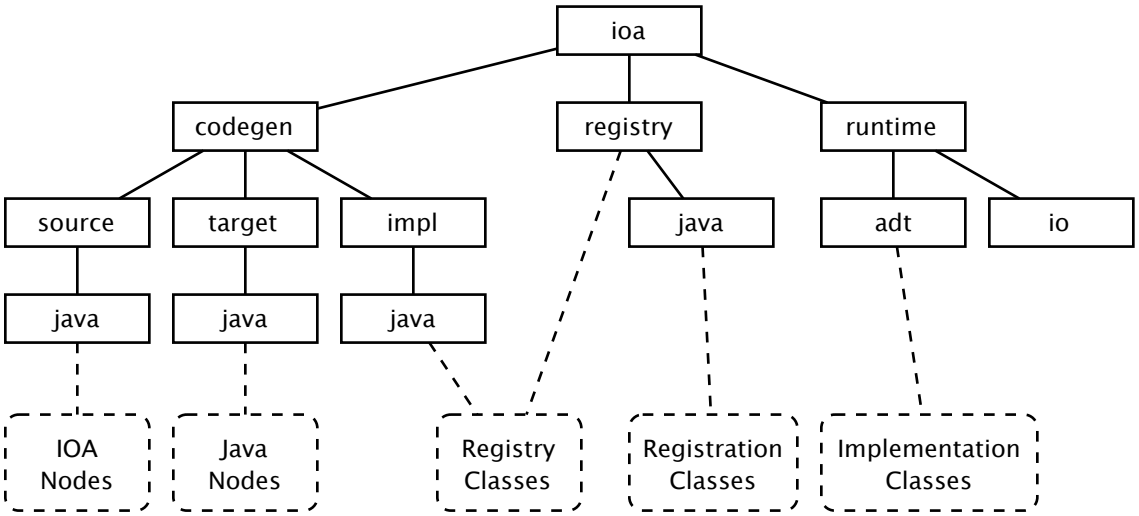

Figure 1: Code Generation Package Structure

The brunt of the code generator's work is in translating the nodes of the source tree into nodes of the target tree. In most cases, this translation proceeds in the same way for all IOA programs: an automaton always translates to a class, a transition always translates to a method, and a state variable always translates to a member variable. These translations can be hard-coded into the nodes of the source syntax tree.

1.2 Data Types Overview

IOA data types are divided into two categories: *sorts* and *sort constructors*. Sorts are simple data types such as integers (Int), real numbers (Real), and booleans (Bool). Sort constructors are compound data types that are parameterized by other sorts or sort constructors. Examples include sequences of integers (Seq[Int]) and mappings from strings to sequences of integers (Map[String, Seq[Int]]). Most of the discussion in this paper applies to both sorts and sort constructors; therefore,

for brevity, I will use "sort" to mean "sort or sort constructor" and clearly indicate sections that apply only to one category of data types.

Data types cannot be translated in the manner of Section 1.1 because they are extensible. At any time, the user can add new data types by specifying them in LSL and implementing them in Java. Doing so should not require modifying the code generator itself.

Each IOA sort is implemented by a Java class, and each operator is implemented by a method on that class. At compile time, one of the code generator's jobs is to map IOA sorts and operators to their Java implementations, so that it can create the proper nodes in the target syntax tree. Much of the work in adding support for new data types involves telling the code generator about this correspondence.

The classes involved in generating code for data types may be divided into three categories:

- *Implementation Classes*, which implement sorts and their operators. These are written by the user, one per data type, and live in `ioa.runtime.ADT`. Of the three kinds of classes, this is the only one that is needed at runtime (i.e. when running the generated code).
- *Registry Classes*, which maintain a mapping between IOA sorts (and operators) and the Java implementation classes (and methods) that implement them. These are built into the code generator.
- *Registration Classes*, which interface between the implementation classes and the registry classes so that each group is isolated from the other. These are written by the user, one per implementation class. They implement the `Registrable` interface and by default live in `ioa.registry.java`.

The relation between these categories and the package structure is shown in Figure 1. The dependencies between them are shown in Figure 2; in particular, note that registration classes decouple the implementation classes from the registry.

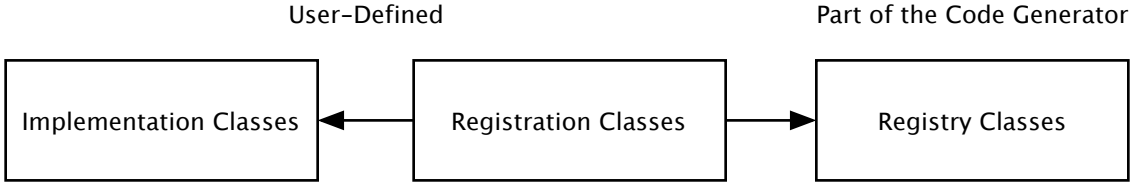

Figure 2: The three types of ADT classes and their dependencies.

1.3 Paper Overview

The remainder of this paper describes how the code generator's ADT classes are designed and how they may be extended to support additional sorts. Section 2 tells how IOA sorts are implemented in Java, Section 3 explains how the code generator uses the registry to map sorts and operators to their implementations, and Section 4 describes how to install new implementation classes into the registry. Section 5 explains the testing architecture for verifying implementation classes, registration classes, and the correspondence between them. Section 6 gives a recipe for writing new ADTs. Section 7 explains how the work described here has been reused in the IOA simulator. Appendixes A and B explain the data types that have already been implemented, and Appendix C contains complete

examples of the files needed to add support for a new data type and examples of the generated Java code.

2 Implementation Classes

The code generator supports the standard sorts defined in the IOA manual[4]: Array, Bool, Char, Int, Map, Mset, Nat, Real, Seq, Set, and String; as well as the shorthand sorts Enum, Tuple, and Union. Each sort is implemented by a Java class called an *implementation class* that belongs to the `ioa.runtime.adt` package. By convention, the name of the implementation class is the name of the sort (e.g., "Seq") followed by "Sort". Thus, booleans are implemented by `BoolSort` and multisets are implemented by `MsetSort`. For brevity, in this section I will sometimes refer to the implementation class as "the ADT."

Each IOA operator is associated with a single sort that introduces it and is implemented by a static method in the implementation class of the introducing sort. For instance, the __+__: Int, Int →Int addition operator on integers is introduced by Int and implemented by `IntSort.add()`. When the code generator generates code for a program, it translates each operator application into a static method invocation. Thus, the static methods form the interface between the implementation class and the rest of the generated code.

2.1 ADTs Extend ioa.runtime.adt.ADT

Implementation classes extend the `ioa.runtime.adt.ADT` abstract class. They must override the non-static `equals()` method inherited from `java.lang.Object`, so that it checks for value equality instead of reference equality.

The `ADT` abstract class provides implementations for two operators that are common to all IOA data types: equality (__=__) and inequality (__~=__)[1] These operators are implemented by static methods in `ADT`. `equals()` and `notEquals()` take two `ADT`s (classes that extend `ADT`) as parameters and return `BoolSort`s that indicate whether the `ADT`s are equal or unequal. The implementations for these operators are automatically installed into the registry; there is no need to mention them in the registration class (see Section 4).

2.2 ADTs are Immutable

With the exception of `ArraySort`, ADTs are immutable. *Author's note: After the next round of modifications, arrays will be normal, immutable ADTs, so you can ignore the following discussion about array limitations.* Each immutable ADT overrides `hashCode()` so that it satisfies the `hashCode()` contract[8]:

> Whenever it is invoked on the same object more than once during an execution of a Java application, the `hashCode()` method must consistently return the same integer. This integer need not remain consistent from one execution of an application to another execution of the same application.
>
> If two objects are equal according to the `equals()` method, then calling the `hashCode()` method on each of the two objects must produce the same integer result.

[1] The conditional operator now shortcuts, so it is no longer implemented in `ADT`.

Since immutable ADTs properly override `hashCode()`, container ADTs may be implemented using hash tables. For instance, `MapSort` uses a hash table to maintain a mapping, and `MsetSort` uses one to map elements to their multiplicities.

This works well except that `ArraySort` is mutable and so cannot satisfy the `hashCode()` contract. Therefore *none of the container ADTs may (correctly) use `ArraySort` as a hash table key*: one cannot create sets of arrays or mappings from arrays to some other type. One can, however, create arrays whose values (not indices) are arrays, which is the standard way to make multi-dimensional arrays in languages such as C.

`ArraySort`'s mutability results in a limitation, which is more general than its inability to satisfy the `hashCode()` contract: there is no way to write a container data structure that organizes itself according to a changeable property of its elements. Even an invariant as simple as the sorted-order of a linked list cannot be maintained if the elements can be mutated without the container's knowledge.

We can envision a contract between element ADTs and container ADTs where the element keeps track of which container it belongs to. Each time it is mutated, it notifies the container, which removes and reinserts the element into the container, thereby ensuring that the container's structure invariants remain satisfied. Such a contract would complicate the design of *all* ADTs for the sake of the single mutable one. Therefore, we do not attempt to solve this problem and instead report violations of the `hashCode()` contract. `ArraySort` overrides its `hashCode()` method to throw an unchecked exception. This way, if the toolset user attempts to use `ArraySort` in a context that requires immutability, she will be alerted to the problem *at runtime*. Unfortunately, we do not know of a way to give the warning sooner.

2.3 Static Methods Implement Operators

The implementation class contains a public static method, also called *code generation methods*, for each of the sort's operators. Making the methods static simplifies code generation by making the syntax of the generated code more regular. However, because static method calls are verbose and unnatural for writing Java data types, all but the simplest implementation classes define two sets of methods: *instance methods* implement the operators on the sort, and *public static methods* are used by the generated code. Implementing the public static methods is simple because they can delegate to the instance methods. Maintaining these two sets of methods is extra work compared to implementing everything using the public static methods; however, it allows for the possibility of easily reusing the implementation classes. Since instance methods are available, hand-coded programs elsewhere in the toolset can use implementation classes as ADTs just as they use `Hashtable` and `Vector`.

The parameters and return values (if any) of the static methods are instances of `ADT`. If the exact type of a parameter is known, then it is specified in the method signature; otherwise, it is labelled `ADT`[2] to indicate that it is known to be an implementation class. The order of the method's parameters is the same as the order of the operands in the domain of the IOA operator. This means that in most cases the first parameter is an instance of the ADT that defines the method[3]. For example, the prototype of the method for the indexing operator on mappings __[__]: Map[D, R], D →R is `public static ADT get(MapSort map, ADT key)`. Its implementation simply calls the `MapSort` instance method: `public ADT get(ADT key)`.

[2] Or `ComparableADT`; see Section 2.7.

[3] An exception is the prepend operator __⊣__: E, Seq[E] →Seq[E].

2.4 Return Value Casting

Java is a statically typed language, and a Java program will only compile if the compiler can verify that methods are called with parameters of the correct types. Container data types are usually written to contain `Object`s, so after taking an `Object` out of a container, one must upcast it (cast it to a more specific type) before using it. This is a general problem with Java containers such as `Vector` and `Hashtable`, and it affects implementation classes in the same way.

To deal with this, whenever the code generator emits a method call, it also emits an upcast for the return value. For instance, the generated code for an IOA term such as head(aSeqOfInt) might be `((IntSort)SeqSort.head(aSeqOfInt_v0))`. The code generator handles this automatically (see Section 3.7), so the implementation class writer need not worry about it. He can assume both that parameters to his classes' methods will be upcast so that they satisfy the method signatures and that non-specific return types such as `ADT` will not pose problems for code using his methods.

2.5 Implicit Initialization and `Parameterizations`

The IOA language allows state variables to be initialized explicitly or implicitly. Explicitly initialized variables are assigned values by the programmer. For instance, an automaton's states section might include the line a: Array[Int, Bool] := constant(true), which initializes a to an array where the value at each integer index is the boolean constant true. Alternatively, the programmer may write a: Array[Int, Bool]. This indicates that the variable is implicitly initialized, and IOA allows the elements of a to take on any values so long as they are of the corrent sort. In this case, the code generator must initialize a before firing any of the automaton's transitions.

When the code generator needs to create an initial value for variable, it first finds the implementation class for the variable's sort and calls its method

```
public static ADT construct(Parameterization p)
```

which is charged with creating and returning "some" instance of the implementation class. The `Parameterization` parameter provides the implementation class with information about the subsorts of the sort that it is implementing. It provides a method `ADT constructSubsort(int)` for creating an instance of the ith subsort of the sort. This is useful when the construction of a sort requires construction of one or more of its subsorts. For instance, `ArraySort.construct(Parameterization)` creates a constant array where the values of all the elements are instances of the element subsort's implementation class.

```
public static ADT construct(Parameterization p)
{
    ADT elementValue = p.constructSubsort(1);
    return constant(elementValue);
}
```

Since `BoolSort.construct(Parameterization)` returns a true `BoolSort`, the above example creates an array whose elements are `BoolSort`s representing true.

Constructing a subsort may involve recursively constructing *its* subsorts. The recursion eventually bottoms out when all the subsorts are unparameterized. In this case, the implementation class's `construct(Parameterization)` method is called, but the ADT ignores the parameter and returns a suitable default (or random) value.

Since the generated code contains `Parameterization` objects, `Parameterization` must be part of the code generator's self-contained runtime package. (See Figure 1.) The implementation registry is *not* part of the runtime, so `Parameterization` may not use the registry when it recursively constructs subsorts. Therefore, code generator looks up all the needed implementation classes at compile time and stores the results in the `Parameterization`. Each `Parameterization` stores the implementation class and `Parameterization` for each of its subsorts.

When the code generator translates an uninitialized state variable, it creates a `Variable` initialized with an `InitialValue` in the target syntax tree. An `InitialValue` is simply a `Term` that stores a `Parameterization` representing its type.

Emitting an `InitialValue` creates a call to the `construct(Parameterization)` method of the implementation class for the value's type, e.g. `ArraySort.construct()` for the above example. The parameter of the method is generated by emitting the `InitialValue`'s stored `Parameterization` object. The return value of `construct()` is cast (from `ADT`) to the appropriate type.

The result of emitting a `Parameterization` is a Java fragment that reconstitutes the `Parameterization`. Again using the above example, the generated code for a: Array[Int, Bool] is:

```
ArraySort a_v0 = (ArraySort)ArraySort.construct(new Parameterization(
                                                  new Class[]
                                                   {ioa.runtime.adt.IntSort.class,
                                                    ioa.runtime.adt.BoolSort.class},
                                                  new Parameterization[]
                                                   {new Parameterization(),
                                                    new Parameterization()}));
```

The parameters to the outermost `Parameterization` constructor are an array of implementation classes for the subsorts and an array of `Parameterization`s for them. The innermost `Parameterization` constructors take no arguments because Int and Bool have no subsorts.

The astute reader will have noticed that `Parameterization` is seemingly a runtime class that knows how to emit itself. This is impossible because knowledge about emitting is confined to the code generator's internals and is not part of the runtime. There are, in fact, two `Parameterization` classes: `ioa.runtime.adt.Parameterization` and `ioa.codegen.target.java.Parameterization`. The former contains the functionality needed at runtime, and the latter (which is a subclass) handles the emitting. Creation of `Parameterization`s is handled by the `ImplFactory`, which in the case of the code generator always creations emittable `Parameterization`s.

2.6 Inheritance

ADTs can inherit method implementations from one their superclass. For instance, `equals()` and `notEquals()` are implemented in `ADT`, and the other ADTs inherit these implementations. When `BoolSort.equals()` is called, `ADT.equals()` is invoked. Inheriting observer methods in this manner will also work for user-defined methods.

In contrast to observer methods, producer methods return new instances of the data type. For instance, `SeqSort.append()` returns a new `SeqSort` based on the original and the parameter. If a subclass of `SeqSort` does not override `append()`, then `append()` will continue to return `SeqSort`s; it will not return instances of the subclass. This is clearly not acceptable because then calling `append()` demotes the subclass to a `SeqSort`, and it *cannot be promoted by casting* because it actually was created as a `SeqSort` and nothing more.

On the other hand, if the subclass is written to override `append()`, then it has gained little from inheritance: just to add a method or modify an existing one, every one of the ADT's producers must be rewritten to return instances of the subclass. And if a producer is added to the base class, then all the derived classes will break. In summary, inheritance does not work as well as we would like, but the limitations lie with immutability rather than with our design of the ADT classes[4].

2.7 Comparing ADTs

Container ADTs such as priority queues require that the elements they contain are totally ordered. To support this, element ADTs that support comparison (such as Int, Real, and String) extend `ioa.runtime.adt.ComparableADT` instead of `ADT`. To do this, they must provide implementations of the following method:

```
public int compareTo(Object object)
```

The parameter is assumed to be a `ComparableADT`, but it is declared as an `Object` so that `ComparableADT` can implement `java.lang.Comparable`. The return value should be zero if the two ADTs are equal, negative if `this` is less than `object`, and positive if `this` is greater than `object`. In ADTs such as `IntSort` that define comparison operators, the comparison operators are implemented in terms of `compareTo()`.

Operators on ordered datatypes are implemented in the normal way, except that the generic datatype is declared as a `ComparableADT` instead of an `ADT`. For example, this method from the priority queue implementation:

```
public static PQSort add(ComparableADT a, PQSort p)
```

Although the priority queue trait assumes that the element sort is totally ordered, the IOA checker does prevent the user from creating priority queues of uncomparable types. In the case of the code generator, such errors will be caught at compile-time when the Java compiler complains that an `ADT` is passed to a method that requires a `ComparableADT`. The same code when run through the simulator results in a `SimException` that reports an illegal method argument.

2.8 Exceptions

The static code generation methods should signal errors and representation violations by throwing `RepExceptions` or subclasses of `RepException`. These unchecked exceptions will be caught by the runtime system. Examples of situations in which it is appropriate to throw a `RepException` are when the code tries to:

- divide by zero
- find the predecessor of the natural number zero
- take the tail of an empty sequence
- pop an empty stack
- access a non-current value of a union

[4] Josh suggests that factories may provide a solution to this problem, but even if they do it is not clear that the extra complexities are justified by the benefits of supporting inheritence.

3 Registry Classes

When the code generator translates a states table or effects clause into Java, it must match IOA sorts and operators to Java classes and methods. To do this, it uses the registry classes, which maintain a mapping between IOA objects and their Java implementations. The principal class that maintains this mapping is `ioa.registry.ConstrImplRegistry`, whose name stands for "Constructor Implementation Registry." I will refer to it simply as "the registry." The contents of the registry are diagrammed in Figure 3.

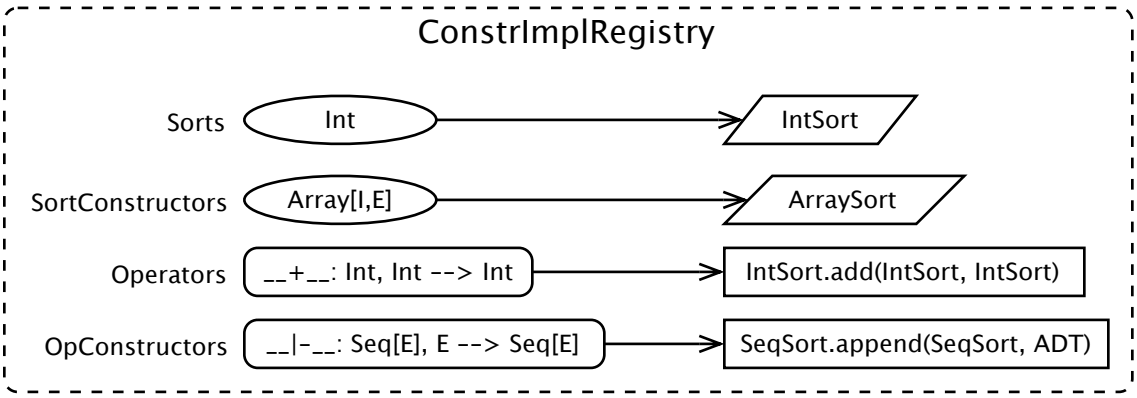

Figure 3: Abstract Contents of the Registry

The registry is used in two phases. In the first phase, the registration classes install their implementation classes into the registry. This process will be described in Section 4. In the second phase, at *compile time*, the code generator uses the registry to look up the implementations of the IOA objects that it needs to emit. This process is diagrammed in Figure 4 and will be described below.

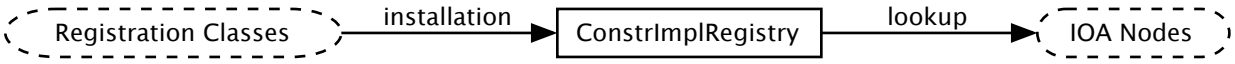

Figure 4: Information Flow and the Registry

To look up a sort or operator, the code generator calls `getImpl()` the registry. This method takes a single parameter, an intermediate language `ioa.il.Sort` or `ioa.il.Operator`, and returns an instance of `ioa.registry.SortImpl` or `ioa.registry.OpImpl`. How the lookup process works and what kinds of `SortImpl`s and `OpImpl`s (which are both subtypes of `Impl`, a marker interface for implementations) are returned depend on whether the sorts and operators are simple or parameterized.

3.1 Looking up Simple Sorts and Operators

For non-parameterized sorts and operators, the registry maintains two tables. The *sort table* maps `Sort` keys to `SortImpl`s, and the *operator table* maps `Operator` keys to `OpImpl`s. Keys are simply structured `String` representations that include the name of the sort or operator and (recursively) all of its subsorts. The syntax of keys is an implementation detail that is subject to change; they are created using the `makeOpKey()` and `makeSortKey()` methods of `ConstrImplRegistry`.

The registry's `installSortImpl()` and `installOpImpl()` methods let one add mappings to the tables, and the `getImpl(Sort)` and `getImpl(Operator)` methods use them to look up `SortImpl` and `OpImpl` objects for simple sorts and operators.

When looking up a simple sort, the registry returns a `SortImpl` that is an instance of `ioa.codegen.target.java.Class`. The `Class` object knows the name of the class that implements the sort (e.g., `ioa.runtime.adt.SeqSort`) and the name of the variable that it represents in the generated code (e.g., `v4`). The code generator calls upon it to emit these pieces of information when the time is right.

Looking up simple operators works in the same way. The registry returns an `OpImpl` that is an instance of `ioa.codegen.target.java.Operator`. The `Operator` is a node in the Java syntax tree, and it knows which method (e.g., `ioa.runtime.adt.SeqSort.head()`) implements the IOA operator that it represents. It also remembers which `ioa.il.Operator` object it is supposed to be implementing. When the code generator asks the `Operator` to emit itself, it passes the `Operator` a list of actual parameters. The `Operator` emits the name of the method (e.g., `SeqSort.head()`) followed by the list of parameter (delimited by commas and enclosed by parentheses). It casts the return value of the method to the implementation class of the sort that the IOA operator returns. This lets one, for instance, take a `StringSort` out of a `SeqSort` and assign it to a variable of type `StringSort`, even though `SeqSort.head()` returns a vanilla `ADT`.

3.2 Looking up Parameterized Sorts and Operators

For parameterized sorts and operators, the registry maintains two more tables. The *sort constructor table* maps `String`s to chains of `SortConstructors`, and the *operator constructor table* maps `String`s to chains of `OpConstructor`s. Unlike their `Impl` counterparts, `SortConstructor` and `OpConstructor` are not nodes of the Java syntax tree of the generated code. Instead, they are intermediary objects that know how to *construct* `Class`es and `Operator`s—that is why they are called "constructors" and extend `ioa.registry.Constructor`.

The keys of the sort constructor table and operator constructor tables are shallow IOA names (omitting subsorts) for sorts and operators (e.g., Map and __⊢__). The values are chains of constructor objects that share the same shallow name. When looking up a parameterized `Sort` or `Operator`, the registry uses the corresponding table to find the appropriate chain of constructors. If the code generator has an implementation for the `Sort` or `Operator`, then one of the `Constructor`s in the chain must claim to be able to implement it. The registry searches down the chain, asking each `Constructor` whether it `match()`es (can implement the given `Sort` or `Operator`; see Section 3.5). When it finds the right `Constructor`, it asks it to construct the appropriate `SortImpl` or `OpImpl`. The registry returns the `Impl`, and from then on everything proceeds as in Section 3.1. Figure 5 shows the proces of generating code for a parameterized operator.

3.3 Curried Parameters

In most cases, there is a one-to-one mapping between IOA operators and the Java methods that implement them. Sometimes, however, it is useful to implement a family of IOA operators with the same Java method. For instance, Char has a zero-ary operator for each character: 'a', 'b', etc. It is convenient to implement all such operators with a single method, `CharSort.lit(char)`. When the IOA program references 'a', the code generator outputs `CharSort.lit('a')`. Here, the `'a'` is a curried parameter that is built-into the subclass of `Operator` object that implements 'a'.

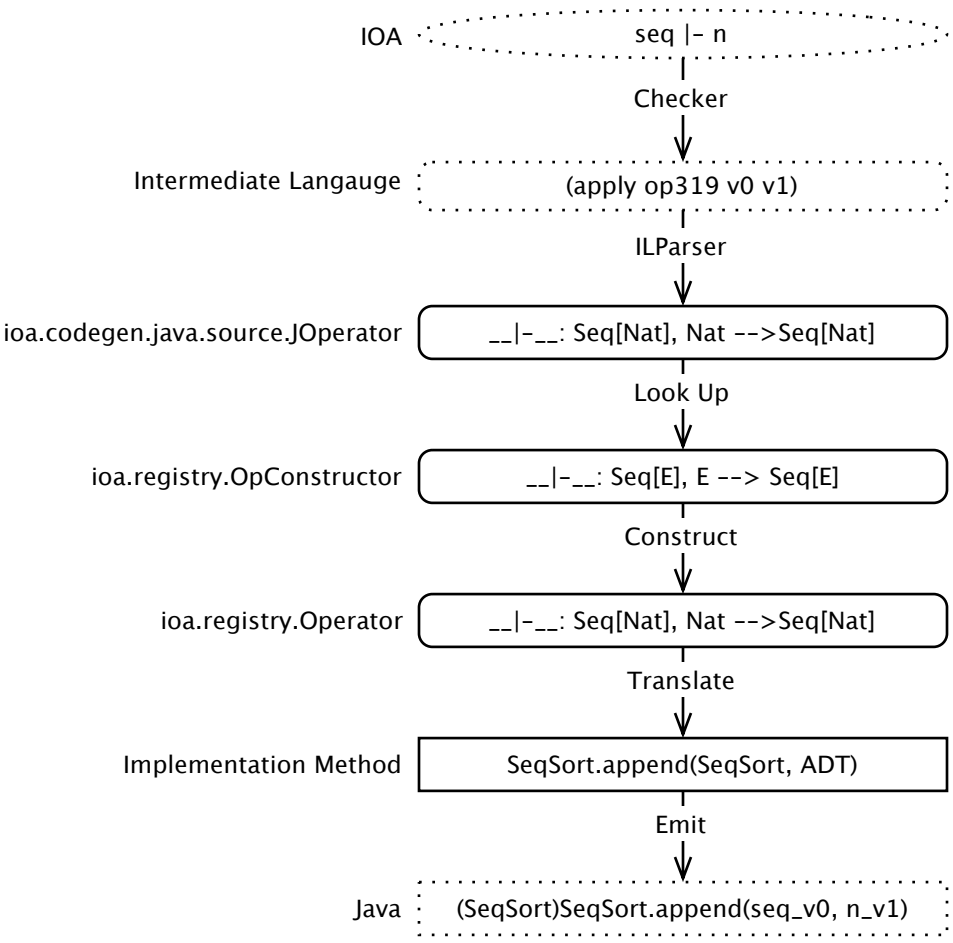

Figure 5: Translating an Operator

This class is called `ExtOperator`[5], and it supports an arbitrary number of curried parameters, which it prepends to the argument list during emission. Operators with curried parameters may be registered using `Installer.installExOpImpl()`; for example, the code from `CharSort` is:

```
// characters is a Collection[Character]
Iterator iter = characters.iterator();
while ( iter.hasNext() )
{
    Character c = (Character)iter.next();
    String opName = "'" + c.charValue() + "'";
    Vector builtIns = new Vector();
    builtIns.add(c);
    installer.installExtOpImpl(opName, name_Char, dom_Empty, "lit", builtIns);
}
```

3.4 Looking up Dynamic Sorts and Operators

In addition to simple sorts and parametrized sorts, there is a third class of sorts called *dynamic sorts*, which includes tuples, unions, and enums. Dynamic sorts are dynamic in the sense that their operators are not known until compile time, when the code generator examines the user-defined data types. Since new shorthand data types may be introduced in each IOA program, the implementation classes for these dynamic sorts must have methods that can each implement a *family* of operators.

For example, a tuple Tup may be defined to have two fields, a and b:

```
type Tupe = tuple of a : Nat, b : Nat}
```

The runtime class for tuples keeps a mapping of field names to field values. One method implements field lookup for all tuples, and another implements field setting for all tuples. These operators are implemented using curried parameters, as described in Section 3.3.

In the Char example from Section 3.3, all the possible characters are known when the *code generator* is compiled; for shorthand sorts, the operators are not known until the *IOA program* is compiled. In the case of tuples, the code generator cannot know *a priori* what the field names are, or even how many fields there are.

The implementations of non-dynamic sorts are installed when the code generator starts up. For dynamic sorts, the implementation classes are installed on demand by the `DynamicRegistry`[6]. *Author's note: What's the technical term for this? It's something like "fault."* If the normal registry cannot find an implementation, it asks the `DynamicRegistry` for one.

When the code generator starts up, all the registration classes for dynamic sorts, *dynamic registration classes*, notify the `DynamicRegistry` of their existence. Then, when the `DynamicRegistry` encounters a `Sort` that the normal registry could not find, it asks each dynamic registration classes if it can install mappings for that sort and its operators. The dynamic registration class, given the `Sort`, then has all the information it requires to do so.

When the `DynamicRegistry` encounters an `Operator` that the normal registry could not find, it first tries to look up the sorts in the `Operator`'s domain. Since one of them must be the sort

[5]The implementation is due to Toh Ne Win.
[6]The implementation is due to Toh Ne Win.

that introduced the operator, looking up the domain sorts will cause the mapping for the operator to be added to the registry; it can then be looked up and returned.

A `Sort` or `Operator` is only looked for in the `DynamicRegistry` once. After that, it will have been added to the normal registry so there will be no need to consult the `DynamicRegistry`.

3.5 Matching Parameterized Sorts and Operators

With parameterized sorts and operators, the table key is a shallow name such as Map or __⊢__ that tells the registry which chain of constructors to search through. Each constructor has a *template*, which indicates the class of sorts or operators that it can implement. From the Javadocs for `ioa.registry.Constructor`, the syntax for templates is defined by the following grammar:

```
<sort>          :: name || (<name> <subsorts>) || <sort variable>
<name>          :: "<id>"
<subsorts>      :: <sort>+
<id>            :: <TERMINALS>
<sort variable> :: <INTEGER>

<op>            :: (<name> domains range)
<domains>       :: (<sort>+)
<range>         :: <sort>
```

Each `SortConstructor` has a template of the form `(<name> <subsorts>)`. For instance, the `SortConstructor` for `MapSort` is `("Map" 0 1)`. Since the sort variables `0` and `1` can each match any sort, this template says that `MapSort` can implement any kind of Map that has two, possibly different, subsorts. This template would match Map[Int, Real], and it would also match Map[Int, Seq[Real]]. If the template had been `("Map" 0 0)`, then the two subsorts would have to be the same; the `SortConstructor` could then match Map[Int, Int] or Map[Seq[Int], Seq[Int]], but not Map[Int, Real]. `OpConstructors` have templates of the form `<op>`, and matching operators to templates works the same as matching sorts to templates.

In general, matching is a recursive process: a sort template matches a sort if it has the same shallow name and its `<subsorts>` can match the sort's subsorts. An operator template matches an operator if it has the same name and if the sorts in its domain and range match the pattern in the template. The recursion must bottom out because the template is of finite length and therefore has finitely many nestings. Mutual recursion is not possible because templates do not contain references, only values.

Presently, the Java code generator uses only a fraction of the flexibility provided by constructors and templates. For instance, all sequences are implemented using `SeqSort`, but the registry classes could support different implementation classes for different operators.

In the future we plan to simply the syntax of templates by removing the need for quotes (and therefore escaping) and adding support for a "me" shorthand that refers to the data type currently being installed.

3.6 Installer

The registry provides four methods for adding mappings to it:

- `installSortImpl(String key, boolean isLiteral, SortImpl impl)`

- `installOpImpl(String key, OpImpl impl)`
- `installSortConstructor(String name, boolean isLiteral, SortConstructor sortCon)`
- `installOpConstructor(String name, OpConstructor opCon)`

Unfortunately, it is cumbersome to use these methods. One must call `ConstrImplRegistry` methods to make operator and sort keys using the registry's static `makeOpKey()` and `makeSortKey()` methods. One must write classes that extend `SortImpl`, `OpImpl`, `SortConstructor`, and `OpConstructor` (which are all abstract) and create instances of them. This can certainly be done, but it needlessly complicates the interface to the registry. Therefore, a *facade*[1] class called `ioa.registry.Installer` mediates between the ADT writer and the registry, providing a clean interface that does not expose to the ADT writer any of the classes that make up the code generator.

One creates an instance of `Installer` using the `ImplFactory` and passing it a reference to the registry, the name of the sort being installed, and the name of the implementation class of that sort. The factory will then return an `Installer` specialized for code generation or simulation. Once an `Installer` has been created, *its* methods may be used to add mappings to the registry. The commonly used methods are:

- `installSortImpl(String implClassPackage, boolean isLiteral)`
- `installOpImpl(String name, String range, String[] domain, String methodName)`
- `installSortConstructor(boolean isLiteral, String template)`
- `installOpConstructor(String opName, String template, String methodName, int numParams)`

And there are also some less commonly used methods to handle special cases:

- `installAssignOpConstructor(String opName, String template, String methodName, int numParams, String assignMethodName)` (See Appendix A.3.)
- `installExtOpImpl(String name, String range, String[] domain, String methodName, Vector builtIns, boolean arrayMode)` (See Section 3.3.)
- `installShortcutOpImpl(String opName, String range, String[] domain, String methodName, String style)` (See Section 3.8.)
- `installShortcutOpConstructor(String opName, String template, String methodName, int numParams, String shortcutStyle)` (See Section 3.8.)

The parameters for these methods are all built-in Java types for which the compiler can recognize literals. Thus, calls to `Installer`'s methods are concise and the programmer using `Installer` is isolated from the inner workings of the code generator.

`Installer` has a simple implementation: it creates the keys and `Impl` objects needed to call the registry methods listed at the top of Section 3.6. A new instance of `Installer` is created for each data type that will be added to the registry. The instance remembers the registry in which to install, the sort being installed, and the implementation class for it. `Installer`'s methods use this information, in addition to their parameters, to create instances of `ioa.codegen.target.Class` (for simple

sorts), `ioa.codegen.target.Operator` (for operators on simple sorts), `ioa.codegen.impl.java.JavaSortConstructor` (for sort constructors), and `ioa.codegen.impl.java.JavaOpConstructor` (for operators on sort constructors). It then installs these instances into the registry. `Class`, `Operator`, `JavaSortConstructor`, and `JavaOpConstructor` are the canonical Java implementations of the `Impl` and `Constructor` objects described in Sections 3.1 and 3.2. Aside from being specialized for emitting Java code, they are augmented with functionality for testing (see Section 5.3).

3.7 Looking up Return Types

As described in Section 3.1, when an `Operator` emits itself as a method call on an implementation class, it also casts the return value to the proper implementation class. Some implementation class methods always return the same type; for instance, the cardinality of a set is always represented by an `IntSort`. Others, however, return different types in different circumstances; for instance, when indexing into an array, the type of the return value depends on how the array is parameterized.

The needed information is available in the source (IOA/IL) syntax tree: each `ioa.il.Operator` stores an `ioa.il.Sort` object that represents its range. Therefore, the registry classes must find this `Sort`, look it up in the registry to find its implementation class, and give the implementation class to the `ioa.codegen.target.java.Operator`, so that it can emit the proper casts.

Ideally, perhaps, the constructor for `Operator` could include a parameter for the implementation class of its return type; that way, every `Operator` would always know how to cast its return value. However, this is not always possible: the registry is populated first by sort A and its operators, then by sort B and its operators, etc. Thus, each operator on A is constructed before sort B is even in the registry—so clearly there is no way to look up the implementation class for B while creating an operator on A. In fact, there can even be cyclic dependencies. For instance, Int and Nat define conversion operators that each depend on the other sort: nat() operates on Ints and returns a Nat, and int() operates on Nats and returns an Int.

Therefore, `Operator`'s constructor does *not* take an implementation class as a parameter. Instead, it must find it after the registry has been fully populated. The code generator has no mechanism at this time for notifying objects that the registry has been fully populated and emission has begun. Therefore, for simplicity, we use the first emission of the `Operator` as a proxy for this notification.

For modularity reasons, `Operator` does not know about the objects in the source syntax tree or about the registry. Therefore, to look up the implementation class of its return type, it uses a helper object called a `ReturnClassThunk`. As its name implies, `ReturnClassThunk` is a thunk that knows how to find the implementation class of the return type of an operator. Each time an `Operator` is created, it is passed a `ReturnClassThunk`, which encapsulates access to the source syntax tree and the registry. When the `Operator` needs to find the return class, it calls the thunk's `lookupReturnClass()` method, which does the actual work.

3.8 Shortcutting

The implementations of the __/__, __\/__, and if__then__else__ operators shortcut. That is, if the first clause of a conjunction is false, the second is not evaluated; if the first clause of a disjunct is true, the second is not evaluated; and only the then or else clause of a conditional is evaluated, depending on the value of the predicate. These behaviors are useful in code generation because they allow the user to guard against runtime exceptions (see Section 2.8).

The above operators cannot be implemented in the usual way because Java always evaluates all method parameters. Instead, they are implemented using `ShortcutOperator`[7], which emits special forms in the target language. For instance, the if__then__else__ operator is implemented using Java's ternary operator:

```
 IOA: a := if (x > y) then a else b

Java: a_v2 = (((BoolSort)IntSort.gt(x_v0, y_v1))).booleanValue() ? a_v2 : b_v3;
```

and the __\/__ operator is implemented using Java's shortcutting `&&` operator:

```
 IOA: c := b /\ a

Java: c_v2 = BoolSort.lit(b_v1.booleanValue() && a_v0.booleanValue());
```

Some glue code is needed to convert between implementation classes and Java's primitive types.

3.9 Locating ADTs at Compile-Time

When the code generator starts up, it calls upon the `ADTLoader` to find registration classes that should be installed in the registry. The `ADTLoader` searches directories and jar files for classes that implement the `Registrable` interface. Search path and exclusions are specified in the `.ioarc` file and may be overridden on the command line. This lets one, for example, choose from alternate implementations of a data type at compile-time. *Author's note: When Atish writes his summer report I'll cite it here.*

4 Registration Classes

Each implementation class in `ioa.runtime.adt` (see Section 2) has a corresponding registration class in `ioa.registry.java`. The implementation class and the registration class have the same name (although this is not depended upon). The job of the registration class is to populate the registry with mappings for the sort and its operators. As such, the registration class depends on the registry and is strongly coupled with its implementation class. Any changes to the specification of an implementation class must be propagated to the corresponding registration class.

Each registration class implements the `ioa.registry.Registrable` interface:

```
public interface Registrable
{
    public void install(ConstrImplRegistry reg) throws RegistryException;
}
```

The code generator calls the `install()` method to populate the registry with mappings from the registration class. In addition, by convention, each registration class includes two constants:

```
public final static String sortName  = "Foo";      // name of the IOA sort
public final static String className = "FooSort";  // name of the implementation class
```

Registration classes use these when they need to refer to other sorts. For instance, the `StringSort` registration class uses `IntSort.sortName` to reference the name of the sort that its len (length) operator returns (Appendix C.1.3).

[7] The implementation is due to Toh Ne Win.

4.1 Standard Registration Classes

The body of the `install()` method first creates an instance of `Installer` (see Section 3.6) by passing it `reg` (which was a parameter of `install()`) and the two constants defined in Section 4. It then uses the `Installer` to add mappings for the sort and all its operators. When installing an operator, the registration class passes to the `Installer` method the name of the operator, its template (see Section 3.5), the name of the implementation class method that implements it, and the number of parameters that the method expects. See Appendixes C.1.3 and C.2.3 for example `install()` methods.

Note that the registration class need not install the equality, inequality, and conditional operators; these are the same for all ADTs and so they are registered automatically by the call to `installSortImpl()` or `installSortConstructor()`. Note also that if the operator name contains a backslash (e.g., the `__/\__` operator on Bools), then the operator name in the string literal passed as the template should contain *four* backslashes[8]. Two of them are removed by the Java compiler when it generates a `String` object from the string literal. This leaves two, which are necessary because the intermediate language s-expression parser expects backslashes to be escaped.

In addition to normal operators, for sorts with literals like Int and Real the registration class installs a special literal operator. The syntax for registering it looks like this:

```
installer.installOpImpl("@<const>lit", "Int", "lit", 1);
```

This tells the code generator that it can use `IntSort.lit()`, which takes an ordinary Java `int`, to create `IntSort`s from integer literals in the source program. In order for this to work, the `isLiteral` should be set to `true` when installing the sort itself.

4.2 Dynamic Registration Classes

When information about a dynamic sort (see Section 3.4) is known, the main registry asks the dynamic registry to actually install the dynamic sort. At this point, the dynamic registry calls upon the dynamic registration class (which extends `DynamicRegistration` and implements `Registrable`) to run its `installDynamic()` method.

`installDynamic()` uses Installer's `installExtOpImpl()` method to add the dynamic sort's operators the registry. `installExtOpImpl()`'s signature is like that of `installOpImpl()`, but it has two additional arguments.

First, there is a `Vector` of extra information, `builtIns`, that contains curried parameters that will be passed to the code generation method when it is called. For example, in the `lookupField()` method of `TupleSort`, the built-in argument is a `Vector` with one argument, the field name. Hence, the `__.a` operator on a tuple would map to `TupleSort.lookupField("a", <tuple>)` and `__.b` would map to `TupleSort.lookupField("b", <tuple>)`.

Second, `installExtOpImpl()` supports an "array mode" where all the arguments to the code generation method are passed as one array. This allows operators with a variable number of arguments. For example, `TupleSort.make()` implements the mixfix `[__,...,__]` operator for creating tuples. `TupleSort.make()` has to take an array of arguments because different tuples have different numbers of fields.

[8] However, the string literal passed as the operator name should contain only two backslashes. After the new s-expression parser is integrated, this inconsistency will go away and both will require two. Eventually, registration classes will become data files and no escaping will be required (because there will be no quoting).

4.3 Non-Standard Registration Classes

A few non-standard operators do not fit the patterns that `Installer` knows about. The non-standard parts must be registered "manually" (as described in the first part of Section 3.6). They must use the `ConstrImplRegistry` instance passed to the registration class to add mappings directly, and they must define their own code emitters (that handle casting if applicable). An example in which a non-standard registration classe is required is `LSeq` (Appendix B.1).

5 Test Classes

5.1 Testing Implementation Classes

Each implementation class has a corresponding *test class*, implemented using the JUnit testing framework[7]. See Appendix C.2.4 for an example. By convention, the test class for `FooSort` is called `FooSortTest`. Test classes live in `ioa.test.junit.runtime.adt` and must extend `TestCase`, which is provided by JUnit. They follow the standard JUnit pattern, containing:

- A `main()` that calls `junit.textui.TestRunner.run(suite())`. This lets one test a particular class by invoking `java` on its test class.
- A `setUp()` method that creates variables (typically instances of ADTs) that will be shared by the test methods. In JUnit terminology, the `setUp()` method creates the *fixture*.
- A `suite()` method whose body is `return new TestSuite(FooSortTest.class);` (where `FooSort` is the name of the ADT being tested). This tells JUnit to create a suite of all the tests for the ADT.
- Test methods for `hashCode()`, `equals()`, and each implementation method. The name of a test method *must* begin with `test`. The rest of the test method name should be the name of the method being tested. (For instance, `BoolSortTest` contains a `testLte()` method that tests `BoolSort`'s `lte()` method.) Test methods consist of a series of calls to `assert()` and `assertEquals()`, which are provided by JUnit. Together, the assertions perform black box and glass box tests.

In addition, each test class must be listed in the `AllADTsTest` class. This can be accomplished by adding a new line to the `suite()` method that says

```
result.addTest(FooSortTest.suite());
```

Once this has been done, one can test the new ADT along with all the old ones by running `java ioa.test.junit.runtime.adt.AllADTsTest`, or by running `make` in the `Test/codegen/java` directory. To test a single ADT, run `java ioa.test.junit.runtime.adt.FooSortTest`.

If an equality assertion fails, JUnit will print out the value it expected and the value it received. The values may then be compared to track down the bug. To take advantage of this feature, ADTs should override `Object.toString()` to "unparse" themselves.

5.2 Testing Registration Classes

It is also important to test the registration class to make sure that the signatures of the operators that it registers match the signatures of the operators that the front end outputs. This is accomplished by creating an IOA program that uses all of the sort's operators (see Appendix C.2.5) and running it through the code generator. These test programs are stored in `IOA_Toolkit/Test/`. Instructions for creating and running the tests are displayed by the `make help` and `make add-help` commands in the directory.

5.3 Catching Bugs in the Implementation/Registration Interface

The tests in Section 5.2 do not ensure that the registration class maps operators to the correct methods, or even to methods that exist. Manual checking must be used to determine the correctness of the mapping, but the `ioa.test.junit.codegen.impl.java.CorrespondenceTest` class can increase confidence in the registration class by checking that the methods it registers actually exist in the implementation class.

`CorrespondenceTest` uses the list of `Registrables` found by the `ADTLocator` to decide which classes to test. It works by posing as a registry to each registration class. When the registration class installs an operator, `CorrespondenceTest` checks that the implementation class includes a suitable function to implement it. In order to do this, it uses the `getMethodName()` and `getNumParameters()` methods of `Operator` or `JavaOpConstructor` to get information about the method being registered. These two method are included in the `JavaMethod` interface, so `CorrespondenceTest` will work with any registered object that implements `JavaMethod`.

One problem with this design is that since `Operator` implements `JavaMethod`, so will any class that extends it (to provide non-standard functionality, for instance); and therefore `CorrespondenceTest` will think that it can test the subclass. In fact, the subclass may be so non-standard that the information provided by the `JavaMethod` accessors no longer makes sense. To handle this situation, `JavaMethod` includes another method, `isTestable()`, that lets test classes determine whether the registered object claims it can be tested as described above. Subclasses of `Operator` that do non-standard things should override `isTestable()` to return `false`, thus preventing spurious errors when `CorrespondenceTest` is run.

6 Recipe for Writing ADTs

To add a new ADT to the code generator, you must:

- Create an implementation class in the `ioa.runtime.adt` package. Make sure that it extends `ioa.runtime.adt.ADT`. Include a public static method for each operator. Be sure to override the non-static `equals()` and `hashCode()` methods from `java.lang.Object`. Also, create a static `construct(Parameterization)` method.
- Create a registration class in the `ioa.registry.java` package. Use `Installer` to install the sort and its operators.
- Write a JUnit-based test class for the implementation class and add it to `AllADTsTest`. Be sure to test `equals()` and `hashCode()`.
- Write an IOA test for the registration class and hook it up to the Makefile-based tester.

Code Generator	Simulator
ioa.codegen.target.java.Class	ioa.simulator.impl.BasicSortImpl
ioa.codegen.impl.java.JavaSortConstructor	ioa.simulator.impl.SimSortConstructor
ioa.codegen.target.java.Operator	ioa.simulator.impl.BasicOpImpl
ioa.codegen.target.java.ShortcutOperator	ioa.simulator.impl.ShortcutOpImpl
ioa.codegen.target.java.ExtOperator	ioa.simulator.impl.ExtOpImpl
ioa.codegen.impl.java.JavaOpConstructor	ioa.simulator.impl.SimOpConstructor
ioa.codegen.impl.java.JavaShortcutOpConstructor	ioa.simulator.impl.SimShortcutOpConstructor

Table 1: Class Correspondence Between the Code Generator and Simulator

To add a new dynamic sort, there are only a few differences:

- Some operators are likely use curried parameters and are registered using `installExtOpImpl()`.
- The `install()` method should simply add a stub instance to `DynamicRegistry`:

  ```
  DynamicRegistry.addDynReg(new TupleSort());
  ```

- The registration class should have a `boolean isDynamic(Sort sort)` method that returns `true` if the given sort can be implemented dynamically by the registration class. `DynamicRegistry` will query this method whenever a dynamic sort is in need of installation.
- The registration class should have an `installDynamic(Sort sort)` method that does the actual installing. Install the necessary operators by calling `installExtOpImpl()` and/or `installOpImpl()` here. `DynamicRegistry` will call this method once `isDynamic()` returns true. Remember to call `installSortImpl()` to install the regular comparison and conditional operators.

7 Sharing ADTs with the Simulator

The abstract data types and infrastructure described in this paper were originally developed for the IOA code generator. Concurrent with the code generator work, [9, 10, 11] were developing a simulator that interprets IOA. The two projects faced similar issues in matching IOA sorts and operators with Java implementations. In fact, the simulator had a registry and its own Java implementations of some basic IOA data types[9].

For obvious reasons, we determined that the code generator and simulator should use the same ADT implementations at runtime. They now use the same implementation classes, the same registration classes, and the same registry. The difference is in the mappings they install into the registry. In each case, the registration class gets an `Installer` from a factory and uses it to populate the registry. For the code generator, the factory returns a `CGInstaller`; for the simulator it returns a `SimInstaller`.

Table 1 shows the correspondence between the classes in the code generator and those in the simulator. Mappings to the classes on the left are installed by `CGInstaller` and mappings to the ones on the right are installed by `SimInstaller`.

[9]The registry worked in a similar manner. The ADT implementations were interwoven with the code that registered them, which made adding new ADTs difficult.

Where code generator classes `emit()`, simulator `*Impl` classes `apply()`. Where `Operator` *emits* code that calls a static implementation class method, `BasicOpImpl` uses Java's reflection API to lookup that method and *run* it. Curried parameters are handled in the same way as in the code generator, and shortcutting operators are similarly special-cased.

A Standard IOA ADTs

With the exception of `ArraySort` (*Author's note: soon to change*), all the IOA ADTs are immutable. Thus, all of the container ADTs are slow because insertion and deletion require an amount of copying that is linear in the number of contained elements. The following ADTs are built into the IOA language.

A.1 BoolSort

Bool is a predefined sort in LSL and IOA, so *every other sort depends on its implementation*, `BoolSort`, written by Joshua A. Tauber. `BoolSort`s are interned: there is only one object for `true` and one for `false`. Other ADT implementations may wish to access these values; for instance, the set membership operator always returns an instance of `BoolSort`. Three public class methods[10] are provided for doing this: `True()`; `False()`; and `lit()`, which converts a Java boolean into a `BoolSort`.

A.2 IntSort

`IntSort` is a straightforward implementation of integers written by Joshua A. Tauber. *Other ADTs that return integers depend on `IntSort`.* For instance, the count operator implemented by `MsetSort` returns `IntSort`s that indicate the multiplicities of elements in the set. To support uses such as this, `IntSort` provides a public class method `lit()`, which converts Java `Integer`s into `IntSort`s. The nat() conversion operator is implemented, but the front end currently does not know how to parse it.

At present, `IntSort` is implemented using `int`. In the future we will have an alternate implementation that uses `java.math.BigInteger` to handle larger integers[11] (for instance, those generated by Fibonacci). A design that I call "internal factories" will allow the user to switch between different Int implementations at runtime.

A.3 ArraySort

`ArraySort` maps indices to values. The space of possible indices is very large—in theory, it is unbounded. Therefore, `ArraySort` uses a sparse representation based on `Hashtable`. Two-dimensional `Array`s are implemented in the same class, as `Hashtable`s mapping pairs of indices to values. They are not yet supported by the intermediate language. When an `ArraySort` is created, it can optionally have a constant value, which acts as a default for indices that are not explicitly mapped in that `Hashtable`.

Indices and values must be immutable, and indices must properly override `equals()` and `hashCode()`.

[10] `True()` and `False()` are capitalized (contrary to our naming convention) to avoid conflicts with the Java keywords `true` and `false`.

[11] `BigInteger` supports integers of up to 2^{32} decimal digits.

In IOA, each element of an array is considered a separate, immutable state variable, and the array notation is merely a shorthand for accessing them. Though statements such as array[i] := j only change the value of individual state variables, they appear to be mutating the array object. Thus, the runtime implements all the elements of an array with a single, mutable `ArraySort` object. *Author's note: This will soon change.*

When translating an assignment node (`ioa.codegen.source.java.JAssignment`) in the source syntax tree, the code generator checks to see if the lvalue is an operator application instead of a variable (the normal case). If so, it creates a special `AssignmentApplicationTerm` for it in the target syntax tree. An `AssignmentApplicationTerm` applies an `AssignOperator`.

Normal `ApplicationTerm`s emit themselves using their `emitApplication()` method; `AssignmentApplicationTerm`s emit themselves using `emitAssignment()`. This method appends the value on the right hand side of the assignment statement to the end of its operand list. Then, it uses the normal machinery to emit an operator application.

The result is a call to `ArraySort.setElementAt()` with three parameters: the `ArraySort` to modify, the index of the element to change, and the new value of the element. Note that an `AssignOperator` does not emit a cast for the return value. This is because the Java compiler will not accept statements of the form:

```
(ArraySort)setElementAt(array, index, value)
```

A.4 CharSort

`CharSort` is a simple wrapper for Java `char`s. It implements lexicographic comparison by comparing Unicode values, which means that `'A'` is less than `'a'`. `CharSort`s are not interned at present, although this would be a simple space and time optimization.

Unlike for integers and reals, the front end does not (yet) specially handle character literals. Instead, each character literal is a zero-ary operator whose range is a Char. These operators are all implemented by the `CharSort.lit()` method and registered with curried parameters, as described in Section 3.3.

A.5 MapSort

`MapSort` is much like `ArraySort`. It differs in that it does not support constant values (because IOA Maps do not), it is immutable, and it is implemented with a `HashMap` from the Java Collections framework. (This is the new way of doing things, and it should be faster than a `Hashtable` because the methods are not synchronized.) Objects in the domain and range must be immutable, and objects in the domain must properly override `equals()` and `hashCode()`.

A.6 MsetSort

`MsetSort` implements IOA multisets using a `HashMap` to map elements (`Object`s) to their multiplicities (`Integer`s). Objects stored in the set must be immutable and must properly override `equals()` and `hashCode()`.

A.7 NatSort

`NatSort` is implemented similarly to `IntSort`. It throws an exception if asked to contain a negative number. The int() conversion operator is implemented but the front end currently does not know

how to parse it. At present, `NatSort` is implemented using `int`; in the future, it may be desirable to instead implement it with `java.math.BigInteger` and internal factories.

A.8 RealSort

`RealSort` implements real numbers using Java `doubles`. At present, it ignores floating point precision issues. As a result, values that should be equal may be reported as unequal (and vice-versa). In the future we should decide whether there should be any sort of conversion between reals and integers. At present, `RealSort` is implemented using `double`; in the future, it may be desirable to instead implement it with `java.math.BigDecimal` and internal factories.

A.9 SeqSort

`SeqSort` is a `Vector`-based implementation of Seq by Joshua A. Tauber.

A.10 SetSort

`SetSort` implements IOA sets using a `HashSet` for speed. Because it was written before `MsetSort`, it is not based on `MsetSort`. Also, it is probably faster this way.

A.11 StringSort

`StringSort` implements String using Java `Strings`.

A.12 EnumSort, TupleSort, and UnionSort

These shorthand sorts were implemented by Toh Ne Win and Laura G. Dean. Because information about user-defined types is not known until after the program has been parsed, these sorts have dynamic registartion classes to create their operator implementations on-demand. See Sections 3.4 and 3.4.

B Other ADTs

User-defined ADTs are defined in external LSL files, which are passed to the `-path` switch of `ioaCheck`.

B.1 LSeqSort

At runtime there are three threads[12]: an input thread run by `ioa.runtime.io.Stdin` appends integers to a shared sequence, stdin; an output thread run by `Stdout` removes integers from a shared sequence, stdout; and the main automaton thread removes integers from stdin and appends (different) integers to stdout.

From an IOA perspective, these sequences are simply Seq[Int] state variables inside a composite automaton with atomic transitions. From a Java perspective, however, the sequence objects are shared between threads. Steps must be taken to ensure that each thread has an up-to-date reference to the shared sequence object and that sections of code that modify these sequences run atomically. Both of these requirements are addressed using locking. The shared sequences are represented

[12]Multiple threads are necessary because the Java platform does not support non-blocking I/O.

using LSeqSorts, which are just like SeqSorts except that they also have support for locking and unlocking.

The Stdin and Stdout classes each have an LSeqSort variable. These are the official references to stdin and stdout. Threads can obtain references to these LSeqSorts by calling Automaton.getStdin() and Automaton.getStdout(). However, because LSeqSort is immutable, they cannot mutate the objects that Stdin and Stdout reference. Therefore, Automaton also provides setStdin() and setStdout() methods that change the references in Stdin and Stdout to point to different objects. As an example, a when the automaton thread removes a value from stdin it follows this sequence of steps (also see Figure 6):

1. The automaton thread obtains a lock on the Stdin class[13]
2. The automaton's variable is updated to point to the global LSeqSort object that represents stdin.
3. The automaton's variable is updated to point to a new LSeqSort object.
4. Stdin's variable is updated to point to the new LSeqSort object.
5. The automaton thread releases the lock on the Stdin class.

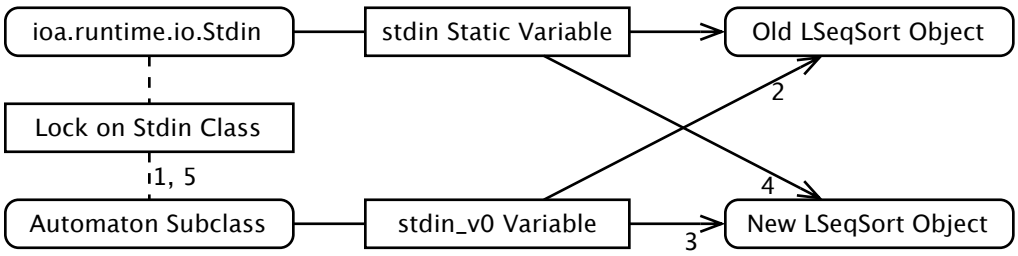

Figure 6: Updating Stdin from the Automaton Thread

The generated Java code to accomplish this is:

```
stdin_v0 = stdin_v0;                                // artifact of translation
synchronized ( ioa.runtime.io.Stdin.class )         // step 1
{
    stdin_v0 = ioa.runtime.Automaton.getStdin();    // step 2
    stdin_v0 = (LSeqSort)LSeqSort.tail(stdin_v0); // step 3
    stdin_v0 = stdin_v0;                            // artifact of translation
    ioa.runtime.Automaton.setStdin(stdin_v0);       // step 4
}                                                   // step 5
```

This process works analogously for the case of modifying stdout. Modification from the Stdin and Stdout threads follows the same general pattern, but that code is built into the runtime, not generated. It will not be discussed here.

[13]It does no good to obtain a lock on an LSeqSort object because it is immutable; each time stdin is changed, Stdin's variable points to a new object. Therefore, we lock on Stdin which, because it is a class, is globally accessible.

The input to the code generator must be a series of IOA statements that translate to the above code. We must add special operators that translate to the `synchronized` block and the code for fetching and updating the shared stdin object. These operators are added as part of the special LSeq sort, which (not surprisingly) is implemented by the above-mentioned `LSeqSort`. LSeq (a lockable sequence) is a special data type that is just like `Seq` with the addition of four new operators: lockStdin, unlockStdin, lockStdout, and unlockStdout.

The IOA code that translates to the above is:

```
stdin := lockStdin(stdin);
stdin := tail(stdin);
stdin := unlockStdin(stdin)
```

The IOA code to obtain a lock on `Stdin` is stdin := lockStdin(stdin). The assignment looks a bit awkward, but it is necessary because IOA does not consider values to be statements. The corresponding Java code for obtaining a lock on stdin is:

```
stdin_v0 = stdin_v0;
synchronized ( ioa.runtime.io.Stdin.class )
{
    stdin_v0 = ioa.runtime.Automaton.getStdin();
```

The first line is a nop that's necessary because of the form of the original IOA statement. The `synchronized` statement establishes a lock on `Stdin`, the input thread class. The input thread cannot append any integers to the sequence while the automaton thread holds the lock on `Stdin`. The third line makes the `stdin_v0` member variable a reference to the shared stdin `LSeqSort` object.

The IOA code to release a lock on `Stdin` is stdin := unlockStdin(stdin). The corresponding Java code is:

```
    stdin_v0 = stdin_v0;
    ioa.runtime.Automaton.setStdin(stdin_v0);
}
```

The first line is a nop. It is there because, as before, the IOA code for releasing a lock must have an lvalue. The call to `setStdin()` updates the shared copy of stdin. Finally, the closing brace ends the `synchronized` block, thus releasing the lock.

Since the locking operators *do not* translate into Java method calls on `LSeqSort`, they need special emitters and a non-standard registration class to install them. The LSeq sort and its standard sequence operators (head, ⊣, etc.) are registered using `Installer` as in Section 4.1. The locking operators are registered as follows. The `LSeqSort` registration class declares static inner classes that extend `LockOp` (a helper class that extends `ioa.codegen.target.java.Operator`), one for each new kind of operator. Each new kind of operator overrides `emitApplication()` to emit itself in a particular way. For example, `LockStdinOp` emits code that establishes a lock on `Stdin` and grabs the shared `LSeqSort` value that represents stdin.

```
public static class LockStdinOp extends LockOp
{
    public LockStdinOp() throws CGException
    {
        super();
    }
```

```
    /**
     * Emit code that will lock Stdin. <TT>opands</TT>
     * must contain a single parameter, the LSeqSort to lock.
     *
     * i.e. if the LSeqSort is v0, then emit:
     *
     * v0;
     * synchronized ( ioa.runtime.io.Stdin.class )
     * {
     *     v0 = ioa.runtime.Automaton.getStdin();
     */
    public Emitter emitApplication(Emitter e, EVector/*[Emittable]*/opands, String op, int numOpands)
        throws CGException
    {
        Emittable stdin = unpackOperand(opands, numOpands);
        e.emit(stdin).put(";\n");
        e.put("synchronized ( ioa.runtime.io.Stdin.class )\n");
        e.put("{\n");
        e.emit(stdin).put(" = ioa.runtime.Automaton.getStdin();\n");
        return e;
    }
}
```

`unpackOperand()` is simply a utility function of `LockOp` that checks the size of the `opands` `EVector` and returns its first element, cast to an `Emittable`. Note that the custom operators do not emit casts for their return values because there are none.

B.2 PQSort

`PQSort` implements priority queues using a binary heap and was written by Atish Dev Nigam. Its elements must be `ComparableADT`s.

B.3 StackSort

`StackSort` implements stacks using a `Vector` and was written by Atish Dev Nigam.

B.4 TreeSort

`TreeSort` implements binary trees and was written by Toh Ne Win.

C Examples

These sections show complete examples of the files one needs to create to add support for a new sort or sort constructor.

C.1 String: A Simple Sort

C.1.1 LSL Trait

This trait simply lists the operators on String; the real LSL trait also states properties of these operators.

```
String: trait
  includes
    Sequence(Char for E, String for Seq[E])
  introduces
```

```
    __<__, __≤__, __>__, __≥__: String, String → Bool
Sequence(E): trait
  includes
    Integer
  introduces
    {}: → Seq[E]
    __⊢__: Seq[E], E → Seq[E]
    __⊣__: E, Seq[E] → Seq[E]
    __‖__: Seq[E], Seq[E] → Seq[E]
    __∈__: E, Seq[E] → Bool
    head, last: Seq[E] → E
    tail, init: Seq[E] → Seq[E]
    len: Seq[E] → Int
    __[__]: Seq[E], Int → E
```

C.1.2 Implementation Class

```
/*
 * Copyright (c) 2001 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.runtime.adt;

/**
 * <tt>StringSort</tt> implements Strings using <tt>java.lang.String</tt>.
 *
 * @author Michael Tsai (00/05/??) -- Wrapper for SeqSort[CharSort]
 * @author Michael Tsai (00/06/27) -- Rewrote to use String
 * @see ioa.registry.java.StringSort
 * @see ioa.runtime.adt.BoolSort
 * @see ioa.runtime.adt.CharSort
 * @see ioa.runtime.adt.IntSort
 */

public class StringSort extends ComparableADT
{
    // Code Generation Methods
    // -----------------------

    /** {}: -> String */
    public static StringSort empty() {
        return new StringSort();
    }

    /** __|-__: String, Char -> String */
    public static StringSort append(StringSort s, CharSort c) {
        return s.append(c);
    }

    /** __-|__: Char, String -> String */
    public static StringSort prepend(CharSort c, StringSort s) {
```

```
    return s.prepend(c);
}

/** __||__: String, String -> String */
public static StringSort catenate(StringSort s1, StringSort s2) {
    return s1.catenate(s2);
}

/** __\\in__: Char, String -> Bool */
public static BoolSort in(CharSort c, StringSort s) {
    return s.in(c);
}

/** head: String -> Char */
public static CharSort head(StringSort s) {
    return s.head();
}

/** last: String -> Char */
public static CharSort last(StringSort s) {
    return s.last();
}

/** tail: String -> String */
public static StringSort tail(StringSort s) {
    return s.tail();
}

/** init: String -> String */
public static StringSort init(StringSort s) {
    return s.init();
}

/** len: String -> Int */
public static IntSort len(StringSort s) {
    return s.len();
}

/** __[__]: String, Int -> Char */
public static CharSort index(StringSort s, IntSort i) {
    return s.index(i);
}

/** __<__: String, String -> Bool */
public static BoolSort lt(StringSort s1, StringSort s2) {
    return s1.lt(s2);
}

/** __<=__: String, String -> Bool */
public static BoolSort lte(StringSort s1, StringSort s2) {
    return s1.lte(s2);
}

/** __>__: String, String -> Bool */
public static BoolSort gt(StringSort s1, StringSort s2) {
    return s1.gt(s2);
}

/** __>=__: String, String -> Bool */
public static BoolSort gte(StringSort s1, StringSort s2) {
    return s1.gte(s2);
}

// Member Variables
// ----------------
protected String string;
```

```
// Creators
// --------
public StringSort() {
    this.string = "";
}

public StringSort(String string) {
    this.string = string;
}

/** @return an instance of StringSort. */
public static ioa.simulator.Entity construct(){
    return empty();
}

public static ADT construct(Parameterization p)
{
    return empty();
}

// Observers
// ---------
public String toString() {
    return this.string;
}

public BoolSort in(CharSort c) {
    return BoolSort.lit(this.string.indexOf(c.toString()) != -1);
}

public IntSort len() {
    return new IntSort(this.string.length());
}

public CharSort index(IntSort i) {
    if ( i.value() < 0 )
        throw new RepException("Index given to StringSort was less than 0");
    else if ( i.value() <= this.string.length() - 1)
        return new CharSort(new Character(this.string.charAt(i.value())));
    else
        throw new RepException("Can't take index "+i+
                               " because the String isn't that long.");
}

public int compareTo(Object o)
{
    StringSort s = (StringSort)o;
    return this.string.compareTo(s.string);
}

public BoolSort lt(StringSort s)
{
    return BoolSort.lit(this.compareTo(s) < 0);
}

public BoolSort lte(StringSort s)
{
    return BoolSort.lit(this.compareTo(s) <= 0);
}

public BoolSort gt(StringSort s)
{
    return BoolSort.lit(this.compareTo(s) > 0);
}

public BoolSort gte(StringSort s)
{
```

```
        return BoolSort.lit(this.compareTo(s) >= 0);
    }

    public boolean equals(Object o) {
        if ( ! (o instanceof StringSort) )
            return false;
        StringSort s = (StringSort)o;
        return this.string.equals(s.string);
    }

    public int hashCode() {
        return this.string.hashCode();
    }

    public boolean isEmpty() {
        return this.string.length() == 0;
    }

    // Producers
    // ---------
    public StringSort append(CharSort c) {
        return new StringSort(this.string + c.toString());
    }

    public StringSort prepend(CharSort c) {
        return new StringSort(c.toString() + this.string);
    }

    public StringSort catenate(StringSort s) {
        return new StringSort(this.string + s.string);
    }

    public CharSort head() {
        if ( ! isEmpty() )
            return CharSort.lit(new Character(this.string.charAt(0)));
        else
            throw new RepException("Attempt to take head() of empty StringSort");
    }

    public CharSort last() {
        if ( ! isEmpty() )
            return CharSort.lit(new Character(this.string.charAt(this.string.length() - 1)));
        else
            throw new RepException("Attempt to take last() of empty StringSort");
    }

    public StringSort tail() {
        if ( ! isEmpty() )
            return new StringSort(this.string.substring(1, this.string.length()));
        else
            throw new RepException("Attempt to take tail() of empty StringSort");
    }

    public StringSort init() {
        if ( ! isEmpty() )
            return new StringSort(this.string.substring(0, this.string.length() - 1));
        else
            throw new RepException("Attempt to take init() of empty StringSort");
    }
}
```

C.1.3 Registration Class

```
/*
 * Copyright (c) 2001 Massachusetts Institute of Technology.
 * All Rights Reserved.
```

```
 *
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.registry.java;

import ioa.registry.ConstrImplRegistry;
import ioa.registry.ImplFactory;
import ioa.registry.Installer;
import ioa.registry.Registrable;
import ioa.registry.RegistryException;

/**
 * Registration class for Strings.
 *
 * @author Michael Tsai
 * @see ioa.runtime.adt.BoolSort
 * @see ioa.runtime.adt.CharSort
 * @see ioa.runtime.adt.IntSort
 * @see ioa.runtime.adt.StringSort
 */
public class StringSort implements Registrable
{
    // Class Variables

    /**
     * Name of the Sort for which this registers implementations.
     */
    public final static String sortName = "String";

    /**
     * Name of the Class that implements the Sort
     */
    public final static String className = "StringSort";

    // Class methods

    /**
     * Install mappings from the sort String and its operators to the
     * ioa.runtime.adt.StringSort class and its methods in the given
     * registry.
     */
    public void install(ConstrImplRegistry reg) throws RegistryException
    {
        final String name_String = sortName;
        final String name_Bool = BoolSort.sortName;
        final String name_Int = IntSort.sortName;
        final String name_Char = CharSort.sortName;
        final String[] dom_Empty = new String[] { };
        final String[] dom_String = new String[] { name_String };
        final String[] dom_String_Char = new String[] { name_String, name_Char };
        final String[] dom_Char_String = new String[] { name_Char, name_String };
        final String[] dom_String_Int = new String[] { name_String, name_Int };
        final String[] dom_String_String = new String[] { name_String, name_String };

        Installer installer = ImplFactory.getInstance().newInstaller(className, sortName, reg);

        // The sort itself (default constructor)
        installer.installSortImpl("ioa.runtime.adt", false);
```

```
        // Empty String operator: {}: -> String
        installer.installOpImpl("{__}", name_String, dom_Empty, "empty");

        // Append element operator:  __|-__: String, Char -> String
        installer.installOpImpl("__|-__", name_String, dom_String_Char, "append");

        // Prepend operator:  __-|__: Char, String -> String
        installer.installOpImpl("__-|__", name_String, dom_Char_String, "prepend");

        // Catentation operator:  __||__: String, String -> String
        installer.installOpImpl("__||__", name_String, dom_String_String, "catenate");

        // Membership operator:  __\in__: Char, String -> Bool
        installer.installOpImpl("__\\in__", name_Bool, dom_Char_String, "in");

        // Head operator: head(__): String -> Char
        installer.installOpImpl("head", name_Char, dom_String, "head");

        // Last operator: last(__): String -> Char
        installer.installOpImpl("last", name_Char, dom_String, "last");

        // Tail operator: tail(__): String -> String
        installer.installOpImpl("tail", name_String, dom_String, "tail");

        // Initial operator: init(__): String -> String
        installer.installOpImpl("init", name_String, dom_String, "init");

        // Length operator: len(__): String -> Int
        installer.installOpImpl("len", name_Int, dom_String, "len");

        // Index operator: __[__]: String, Int -> Char
        installer.installOpImpl("__[__]", name_Char, dom_String_Int, "index");

        // Less than operator: __<__: String, String -> Bool
        installer.installOpImpl("__<__", name_Bool, dom_String_String, "lt");

        // Less than or equal operator: __<=__: String, String -> Bool
        installer.installOpImpl("__<=__", name_Bool, dom_String_String, "lte");

        // Greater than operator: __>__: String, String -> Bool
        installer.installOpImpl("__>__", name_Bool, dom_String_String, "gt");

        // Greater than or equal operator: __>=__: String, String -> Bool
        installer.installOpImpl("__>=__", name_Bool, dom_String_String, "gte");
    }
}
```

C.1.4 Test Class

```
/*
 * Copyright (c) 2000 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.test.junit.runtime.adt;
```

```
import junit.framework.*;
import ioa.runtime.adt.BoolSort;
import ioa.runtime.adt.CharSort;
import ioa.runtime.adt.IntSort;
import ioa.runtime.adt.StringSort;
import ioa.runtime.adt.RepException;

/**
 * JUnit-based black box and glass box tests for ioa.runtime.adt.StringSort.
 * @author Michael J. Tsai (00/06/27)
 */
public class StringSortTest extends TestCase
{
    // Member Variables
    // ----------------
    protected StringSort empty;
    protected StringSort aString;
    protected StringSort bString;
    protected StringSort ab;
    protected StringSort abc;
    protected StringSort abcd;
    protected StringSort bc;
    protected StringSort bcd;
    protected StringSort cd;

    protected IntSort zero;
    protected IntSort one;
    protected IntSort two;
    protected IntSort three;
    protected IntSort minusOne;

    protected CharSort a;
    protected CharSort b;
    protected CharSort c;
    protected CharSort d;

    protected BoolSort bTrue;
    protected BoolSort bFalse;

    /**
     * Runs all the tests in this class and outputs the results to stdout.
     */
    public static void main(String[] args)
    {
        junit.textui.TestRunner.run(suite());
    }

    // Framework Stuff
    // ---------------
    public StringSortTest(String name)
    {
        super(name);
    }

    /**
     * Set up the fixtures.
     */
    protected void setUp()
    {
        this.empty = new StringSort("");
        this.aString = new StringSort("a");
        this.bString = new StringSort("b");
        this.ab = new StringSort("ab");
        this.abc = new StringSort("abc");
        this.abcd = new StringSort("abcd");
        this.bc = new StringSort("bc");
```

```
        this.bcd = new StringSort("bcd");
        this.cd = new StringSort("cd");

        this.a = CharSort.lit(new Character('a'));
        this.b = CharSort.lit(new Character('b'));
        this.c = CharSort.lit(new Character('c'));
        this.d = CharSort.lit(new Character('d'));

        this.zero = new IntSort(0);
        this.one = new IntSort(1);
        this.two = new IntSort(2);
        this.three = new IntSort(3);
        this.minusOne = new IntSort(-1);

        this.bFalse = BoolSort.False();
        this.bTrue = BoolSort.True();
    }

    /**
     * @return a single Test that runs all the tests in this class
     */
    public static Test suite()
    {
        return new TestSuite(StringSortTest.class);
    }

    // Test Methods
    // ------------
    public void testEmpty()
    {
        assertEquals("", empty.toString());
    }

    public void testAppend()
    {
        assertEquals(abc, ab.append(c));
        assertEquals(abcd, abc.append(d));
        assertEquals("a", empty.append(a).toString());
    }

    public void testPrepend()
    {
        assertEquals(abc, bc.prepend(a));
        assertEquals(abcd, bcd.prepend(a));
        assertEquals("a", empty.prepend(a).toString());
    }

    public void testCatenate()
    {
        assertEquals(empty, empty.catenate(empty));
        assertEquals(ab, empty.catenate(ab));
        assertEquals(ab, ab.catenate(empty));
        assertEquals(abcd, ab.catenate(cd));
    }

    public void testIn()
    {
        assertEquals(bTrue, ab.in(a));
        assertEquals(bTrue, abc.in(a));
        assertEquals(bTrue, ab.in(b));
        assertEquals(bTrue, abc.in(b));
        assertEquals(bFalse, ab.in(c));
        assertEquals(bTrue, abc.in(c));
        assertEquals(bFalse, ab.in(c));
        assertEquals(bFalse, empty.in(a));
    }
```

```
public void testHead()
{
    assertEquals(a, ab.head());
    assertEquals(a, abc.head());
    assertEquals(b, bc.head());
    try { empty.head(); fail(); } catch ( RepException e ) {}
}

public void testLast()
{
    assertEquals(b, ab.last());
    assertEquals(c, abc.last());
    assertEquals(c, bc.last());
    try { empty.last(); fail(); } catch ( RepException e ) {}
}

public void testTail()
{
    try { empty.tail(); fail(); } catch ( RepException e ) {}
    assertEquals(empty, aString.tail());
    assertEquals(bString, ab.tail());
    assertEquals(bc, abc.tail());
    assertEquals(bcd, abcd.tail());
}

public void testInit()
{
    try { empty.init(); fail(); } catch ( RepException e ) {}
    assertEquals(empty, aString.init());
    assertEquals(aString, ab.init());
    assertEquals(ab, abc.init());
    assertEquals(abc, abcd.init());
}

public void testLen()
{
    assertEquals(zero, empty.len());
    assertEquals(one, aString.len());
    assertEquals(two, ab.len());
}

public void testIndex()
{
    try { empty.index(zero); fail(); } catch ( RepException e ) {}
    try { aString.index(one); fail(); } catch ( RepException e ) {}
    try { aString.index(minusOne); fail(); } catch ( RepException e ) {}
    assertEquals(a, aString.index(zero));
    assertEquals(b, abc.index(one));
}

public void testLt()
{
    assertEquals(bTrue, aString.lt(ab));
    assertEquals(bFalse, ab.lt(aString));
    assertEquals(bTrue, ab.lt(bc));
    assertEquals(bFalse, bc.lt(ab));
    assertEquals(bFalse, ab.lt(ab));
    assertEquals(bTrue, empty.lt(ab));
}

public void testLte()
{
    assertEquals(bTrue, aString.lte(ab));
    assertEquals(bFalse, ab.lte(aString));
    assertEquals(bTrue, ab.lte(bc));
    assertEquals(bFalse, bc.lte(ab));
    assertEquals(bTrue, ab.lte(ab));
```

```
        assertEquals(bTrue, empty.lte(ab));
    }

    public void testGt()
    {
        assertEquals(bFalse, aString.gt(ab));
        assertEquals(bTrue, ab.gt(aString));
        assertEquals(bFalse, ab.gt(bc));
        assertEquals(bTrue, bc.gt(ab));
        assertEquals(bFalse, ab.gt(ab));
        assertEquals(bFalse, empty.gt(ab));
    }

    public void testGte()
    {
        assertEquals(bFalse, aString.gte(ab));
        assertEquals(bTrue, ab.gte(aString));
        assertEquals(bFalse, ab.gte(bc));
        assertEquals(bTrue, bc.gte(ab));
        assertEquals(bTrue, ab.gte(ab));
    }

    public void testEquals()
    {
        assertEquals(bTrue, StringSort.equals(abc, new StringSort("abc")));
        assertEquals(bFalse, StringSort.equals(abc, bc));
    }

    public void testNotEquals()
    {
        assertEquals(bFalse, StringSort.notEquals(abc, new StringSort("abc")));
        assertEquals(bTrue, StringSort.notEquals(abc, bc));
    }

    public void testIfThenElse()
    {
        assertEquals(abc, StringSort.ifThenElse(bTrue, abc, bc));
        assertEquals(bc, StringSort.ifThenElse(bFalse, abc, bc));
    }

    public void testHashCode()
    {
         assertEquals(ab.hashCode(), aString.catenate(bString).hashCode());
         assert(ab.hashCode() != aString.hashCode());
    }

    public void testCompareTo()
    {
        assert( ab.compareTo(bc) < 0 );
        assert( abc.compareTo(aString) > 0);
        assert( abc.compareTo(abc) == 0);
    }
}
```

C.1.5 IOA File

```
automaton String01
  signature
    internal a1
    states
```

```
    s: String := {},
    i: Int,
    c: Char := 'A',
    b: Bool
  transitions
    internal a1
      pre
        s = {}
      eff
        s := s ⊢ c;
        s := c ⊣ s;
        s := s ‖ s;
        b := c ∈ s;
        c := head(s);
        c := last(s);
        s := tail(s);
        s := init(s);
        i := len(s);
        c := s[2];
        b := s = s;
        b := s[1] = s[2];
        b := s < s;
        b := s ≤ s;
        b := s > s;
        b := s ≥ s;
        b := s = s;
        b := s ≠ s;
        s := if b then s else s
```

C.1.6 Generated Java Code

```
package ioa.runtime;

import java.io.*; import ioa.runtime.adt.*;

public class String01 extends ioa.runtime.Automaton {
  StringSort s_v1 = ((StringSort)StringSort.empty());
  IntSort i_v2 = (IntSort)IntSort.construct(new Parameterization());
  CharSort c_v3 = ((CharSort)CharSort.lit('A'));
  BoolSort b_v4 = (BoolSort)BoolSort.construct(new Parameterization());
  public void a0()
    {
      s_v1 = ((StringSort)StringSort.append(s_v1, c_v3));
      s_v1 = ((StringSort)StringSort.prepend(c_v3, s_v1));
      s_v1 = ((StringSort)StringSort.catenate(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.in(c_v3, s_v1));
      c_v3 = ((CharSort)StringSort.head(s_v1));
      c_v3 = ((CharSort)StringSort.last(s_v1));
      s_v1 = ((StringSort)StringSort.tail(s_v1));
      s_v1 = ((StringSort)StringSort.init(s_v1));
      i_v2 = ((IntSort)StringSort.len(s_v1));
      c_v3 = ((CharSort)StringSort.index(s_v1, ((IntSort)IntSort.lit(2))));
      b_v4 = ((BoolSort)StringSort.equals(s_v1, s_v1));
      b_v4 =
        ((BoolSort)CharSort.equals(((CharSort)StringSort.index(s_v1,
                                                    ((IntSort)IntSort.lit(1)))),
                                   ((CharSort)StringSort.index(s_v1,
                                                    ((IntSort)IntSort.lit(2))))));
```

```
      b_v4 = ((BoolSort)StringSort.lt(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.lte(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.gt(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.gte(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.equals(s_v1, s_v1));
      b_v4 = ((BoolSort)StringSort.notEquals(s_v1, s_v1));
      s_v1 = ((StringSort)((b_v4).booleanValue() ? s_v1 : s_v1));

  }

  public static void main(String[] args) {
ioa.runtime.Automaton.main(new String[] {"String01"});
}

}
```

C.2 Set: A Parameterized Sort

C.2.1 LSL Trait

This trait simply lists the operators on Set; the real LSL trait[14] also states properties of these operators.

```
Set(E): trait
  includes
    Integer
  introduces
    {}: → Set[E]
    {__}: E → Set[E]
    insert, delete: E, Set[E] → Set[E]
    __∈__: E, Set[E] → Bool
    __∪__, __∩__, __-__: Set[E], Set[E] → Set[E]
    __⊂__, __⊃__, __⊆__, __⊇__: Set[E], Set[E] → Bool
    size: Set[E] → Int
```

C.2.2 Implementation Class

```
/*
 * Copyright (c) 2001 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.runtime.adt;

import ioa.util.logger.IOACategory;
import ioa.util.ToStringComparator;

import java.util.Collections;
```

[14] The real trait is not included here because it assumes other traits, which makes it harder to see what all the operators are.

```
import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;
import java.util.TreeSet;

/**
 * <p> <tt>SetSort</tt> implements IOA sets using a <tt>HashSet</tt> for
 * speed. Because it was written before <tt>MsetSort</tt>, it is not
 * based on <tt>MsetSort</tt>. Also, it is probably faster this way.</p>
 *
 * @author Michael Tsai (00/04/19)
 * @see ioa.registry.java.SetSort
 * @see ioa.runtime.adt.IntSort
 * @see ioa.runtime.adt.BoolSort
 */
public class SetSort extends ADT implements Cloneable
{
    // Class Variables
    private static IOACategory cat = IOACategory.getInstance (SetSort.class.getName());

    // Member Variables
    // ----------------

    protected Set set = new HashSet();

    // Creators
    // --------

    /** Construct a new, empty set. */
    public SetSort() {}

    // Code Generation Methods
    // -----------------------

    /** {}:  -> Set[E] */
    public static SetSort empty() {
        return new SetSort();
    }

    /** {__}: E -> Set[E] */
    public static SetSort singleton(Object o) {
        return new SetSort().insert(o);
    }

    /** insert: E, Set[E] -> Set[E] */
    public static SetSort insert(Object o, SetSort s) {
        return s.insert(o);
    }

    /** delete; E, Set[E] -> Set[E] */
    public static SetSort delete(Object o, SetSort s) {
        return s.delete(o);
    }

    /** __\in__: E, Set[E] -> Bool */
    public static BoolSort isIn(Object o, SetSort s) {
        return BoolSort.lit(s.contains(o));
    }

    /** __\U__: Set[E], Set[E] -> Set[E] */
    public static SetSort union(SetSort s1, SetSort s2) {
        return s1.union(s2);
    }

    /** __\I__: Set[E], Set[E] -> Set[E] */
    public static SetSort intersection(SetSort s1, SetSort s2) {
        return s1.intersection(s2);
```

```
}

/** __-__: Set[E], Set[E] -> Set[E] */
public static SetSort difference(SetSort s1, SetSort s2) {
    return s1.difference(s2);
}

/** __\supset__: Set[E], Set[E] -> Bool */
public static BoolSort isSupset(SetSort s1, SetSort s2) {
    return BoolSort.lit(s1.isSupset(s2));
}

/** __\subset__: Set[E], Set[E] -> Bool */
public static BoolSort isSubset(SetSort s1, SetSort s2) {
    return BoolSort.lit(s1.isSubset(s2));
}

/** __\subseteq__: Set[E], Set[E] -> Bool */
public static BoolSort isSubsetEq(SetSort s1, SetSort s2) {
    return BoolSort.lit(s1.isSubsetEq(s2));
}

/** __\supseteq__: Set[E], Set[E] -> Bool */
public static BoolSort isSupsetEq(SetSort s1, SetSort s2) {
    return BoolSort.lit(s1.isSupsetEq(s2));
}

/** size: Set[E] -> Int */
public static IntSort size(SetSort s) {
    return new IntSort(s.size());
}

// Observers
// ---------

/** @return true if this contains o and false otherwise */
public boolean contains(Object o) {
    return set.contains(o);
}

/** @return true if this \subset s and false otherwise */
public boolean isSubset(SetSort s) {
    return s.set.containsAll(this.set) && ! this.set.equals(s.set);
}

/** @return true if this \supset s and false otherwise */
public boolean isSupset(SetSort s) {
    return this.set.containsAll(s.set) && ! this.set.equals(s.set);
}

/** @return true if this \subseteq s and false otherwise */
public boolean isSubsetEq(SetSort s) {
    return s.set.containsAll(this.set);
}

/** @return true if this \supseteq s and false otherwise */
public boolean isSupsetEq(SetSort s) {
    return this.set.containsAll(s.set);
}

/** @return |this| */
public int size() {
    return set.size();
}

/**
 * Returns the underlying java.util.Set, but
```

```
 * the elements are unmodifiable.  The elements
 * are also ordered alphabetically as they'd print out.
 **/

public Set getSet() {
  TreeSet ts = new TreeSet (new ToStringComparator());
  ts.addAll (set);
  return Collections.unmodifiableSet(ts);
}

public boolean equals(Object o) {
    if ( o instanceof SetSort )
        return this.set.equals(((SetSort) o).set);
    else
        return false;
}

public int hashCode() {
    int result = 0;

    Iterator iter = this.set.iterator();
    while ( iter.hasNext() )
    {
        result += iter.next().hashCode();
    }

    return result;
}

// Producers
// --------

/** @return shallow copy of this */
public Object clone() {
    SetSort result = new SetSort();
    result.set.addAll(this.set);
    return result;
}

/** @return this \U {o} */
public SetSort insert(Object o) {
    SetSort result = (SetSort)this.clone();
    result.set.add(o);
    return result;
}

/** @return this - {o} */
public SetSort delete(Object o) {
    SetSort result = (SetSort)this.clone();
    result.set.remove(o);
    return result;
}

/** @return a new SetSort whose elements are this \U s */
public SetSort union(SetSort s) {
    SetSort result = (SetSort)this.clone();
    result.set.addAll(s.set);
    return result;
}

/** @return a new SetSort whose elements are this \I s */
public SetSort intersection(SetSort s) {
    SetSort result = (SetSort)this.clone();
    result.set.retainAll(s.set);
    return result;
}
```

```
    /** @return a new SetSort whose elements are this - s */
    public SetSort difference(SetSort s) {
        SetSort result = (SetSort)this.clone();
        result.set.removeAll(s.set);
        return result;
    }

    /**
     * Select a random element from a set.
     * @return a random element of this set.
     * @exception RepException if the set is empty
     **/
    public static Object chooseRandom (SetSort set) {
        int i = set.size();
if (i == 0) throw new RepException ("Cannot take an element out of an empty set");
return set.getSet().toArray()[NonDet.rnd.nextInt (i)];
    }

    /**
     * Exclude a random element from a set and return the rest.
     * @return a subset of the set such that there's one element removed.
     * @exception RepException if the set is empty
     **/
    public static SetSort rest (SetSort set) {
        int i = set.size();
if (i == 0) throw new RepException ("Cannot take an element out of an empty set");
return SetSort.delete (set.getSet().toArray()[NonDet.rnd.nextInt (i)], set);
    }

    /**
     * Predicate on whether a set is empty
     * @return true of the set is empty.
     **/
    public static BoolSort isEmpty (SetSort set) {
        int i = set.size();
return BoolSort.lit (i == 0);
    }

    /** @return an instance of SetSort. */
    public static ioa.simulator.Entity construct() {
        return empty();
    }

    /** @return an instance of SetSort */
    public static ADT construct(Parameterization p)
    {
        return empty();
    }

    /**
     * Returns a String representation of this, in alphabetical order
     * @return a String representation of this.
     **/
    public String toString() {
        TreeSet ts = new TreeSet();
        Iterator iter = getSet().iterator();
String result = "(";

        while ( iter.hasNext() ) {
            result += iter.next().toString() + " ";
        }
        return result.trim() + ")";
    }
}
```

C.2.3 Registration Class

```
/*
 * Copyright (c) 2001 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.registry.java;

import ioa.registry.ConstrImplRegistry;
import ioa.registry.ImplFactory;
import ioa.registry.Installer;
import ioa.registry.Registrable;
import ioa.registry.RegistryException;

/**
 * Registration class for Sets. The implementations of
 * ioa.registry.java.SetSort and ioa.runtime.adt.SetSort are
 * intertwined. Changes in one should be reflected in the other.
 *
 * @author Michael Tsai (00/04/20)
 * @see ioa.runtime.adt.SetSort
 */
public class SetSort implements Registrable
{
    // Class Variables
    // ---------------

    /**
     * Name of the Sort for which this registers implementations.
     */
    public final static String sortName = "Set";

    /**
     * Name of the Class that implements the Sort
     */
    public final static String className = "SetSort";


    // Class methods
    // -------------

    /**
     * Install mappings from the Sort constructor Set and its operators
     * to the ioa.runtime.adt.SetSort class and its methods in the given
     * registry.
     */
    public void install(ConstrImplRegistry reg) throws RegistryException
    {
        Installer installer = ImplFactory.getInstance().newInstaller(className, sortName, reg);

        // The sort itself (default constructor)
        installer.installSortConstructor(false, "(\"Set\" 0)");

        // {}: -> Set[E]
        // template: ("{__}" () ("Set" 0))
        installer.installOpConstructor("{__}", "(\"{__}\" () (\"Set\" 0))", "empty", 0);
```

```
        // {__}: E -> Set[E]
        // template: ("{__}" (0) ("Set" 0))
        installer.installOpConstructor("{__}", "(\"{__}\" (0) (\"Set\" 0))", "singleton", 1);

        // insert: E, Set[E] -> Set[E]
        // template: ("insert" (0 ("Set" 0)) ("Set" 0))
        installer.installOpConstructor("insert", "(\"insert\" (0 (\"Set\" 0)) (\"Set\" 0))", "insert", 2);

        // delete: E, Set[E] -> Set[E]
        // template: ("delete" (0 ("Set" 0)) ("Set" 0))
        installer.installOpConstructor("delete", "(\"delete\" (0 (\"Set\" 0)) (\"Set\" 0))", "delete", 2);

        // __\in__: E, Set[E] -> Bool
        // template: ("__\\in__" (0 ("Set" 0)) "Bool")
        installer.installOpConstructor("__\\in__", "(\"__\\\\in__\" (0 (\"Set\" 0)) \"Bool\")", "isIn", 2);

        // __\U__: Set[E], Set[E] -> Set[E]
        // template: ("__\\U__" (("Set" 0) ("Set" 0)) ("Set" 0))
        installer.installOpConstructor("__\\U__",
        "(\"__\\\\U__\" ((\"Set\" 0) (\"Set\" 0)) (\"Set\" 0))", "union", 2);

        // __\I__: Set[E], Set[E] -> Set[E]
        // template: ("__\\I__" (("Set" 0) ("Set" 0)) ("Set" 0))
        installer.installOpConstructor("__\\I__",
        "(\"__\\\\I__\" ((\"Set\" 0) (\"Set\" 0)) (\"Set\" 0))", "intersection", 2);

        // __-__: Set[E], Set[E] -> Set[E]
        // template: ("__-__" (("Set" 0) ("Set" 0)) ("Set" 0))
        installer.installOpConstructor("__-__",
        "(\"__-__\" ((\"Set\" 0) (\"Set\" 0)) (\"Set\" 0))", "difference", 2);

        // __\subset__: Set[E], Set[E] -> Bool
        // template: ("__\\subset__" (("Set" 0) ("Set" 0)) "Bool")
        installer.installOpConstructor("__\\subset__",
        "(\"__\\\\subset__\" ((\"Set\" 0) (\"Set\" 0)) \"Bool\")", "isSubset", 2);

        // __\supset__: Set[E], Set[E] -> Bool
        // template: ("__\\supset__" (("Set" 0) ("Set" 0)) "Bool")
        installer.installOpConstructor("__\\supset__",
        "(\"__\\\\supset__\" ((\"Set\" 0) (\"Set\" 0)) \"Bool\")", "isSupset", 2);

        // __\subseteq__: Set[E], Set[E] -> Bool
        // template: ("__\\subseteq__" (("Set" 0) ("Set" 0)) "Bool")
        installer.installOpConstructor("__\\subseteq__",
        "(\"__\\\\subseteq__\" ((\"Set\" 0) (\"Set\" 0)) \"Bool\")", "isSubsetEq", 2);

        // __\supseteq__: Set[E], Set[E] -> Bool
        // template: ("__\\supseteq__" (("Set" 0) ("Set" 0)) "Bool")
        installer.installOpConstructor("__\\supseteq__",
        "(\"__\\\\supseteq__\" ((\"Set\" 0) (\"Set\" 0)) \"Bool\")", "isSupsetEq", 2);

        // size: Set[E] -> Int
        // template: ("size" (("Set" 0)) "Int")
        installer.installOpConstructor("size", "(\"size\" ((\"Set\" 0)) \"Int\")", "size", 1);

// Set[E] -> E
// template: ("chooseRandom" (("Set" 0)) 0)
        installer.installOpConstructor("chooseRandom", "(\"chooseRandom\" ((\"Set\" 0)) 0)", "chooseRandom", 1);


// Set[E] -> Set[E]
// template: ("rest" (("Set" 0)) ("Set" 0))
        installer.installOpConstructor("rest", "(\"rest\" ((\"Set\" 0)) (\"Set\" 0))", "rest", 1);

// Set[E] -> Bool
// template: ("isEmpty" (("Set" 0)) "Bool")
```

```
        installer.installOpConstructor("isEmpty", "(\"isEmpty\" ((\"Set\" 0)) \"Bool\")", "isEmpty", 1);


    }
}
```

C.2.4 Test Class

```
/*
 * Copyright (c) 2000 Massachusetts Institute of Technology.
 * All Rights Reserved.
 *
 * MIT grants permission to use, copy, modify, and distribute this software and
 * its documentation for NON-COMMERCIAL purposes and without fee, provided that
 * this copyright notice appears in all copies.
 *
 * MIT provides this software "as is," without representations or warranties of
 * any kind, either expressed or implied, including but not limited to the
 * implied warranties of merchantability, fitness for a particular purpose, and
 * noninfringement.  MIT shall not be liable for any damages arising from any
 * use of this software.
 */

package ioa.test.junit.runtime.adt;

import junit.framework.*;
import ioa.runtime.adt.SetSort;
import ioa.runtime.adt.BoolSort;

/**
 * JUnit-based black box and glass box tests for ioa.runtime.adt.SetSort.
 * @author Michael J. Tsai (00/06/22)
 */
public class SetSortTest extends TestCase
{
    // Member Variables
    // ----------------
    protected SetSort empty;
    protected SetSort one;
    protected SetSort two;
    protected SetSort three;
    protected SetSort oneAndTwo;
    protected SetSort oneAndThree;
    protected SetSort twoAndThree;
    protected SetSort oneAndTwoAndThree;

    protected BoolSort bTrue;
    protected BoolSort bFalse;

    /**
     * Runs all the tests in this class and outputs the results to stdout.
     */
    public static void main(String[] args)
    {
        junit.textui.TestRunner.run(suite());
    }

    // Framework Stuff
    // ---------------
    public SetSortTest(String name)
    {
        super(name);
    }
```

```
/**
 * Set up the fixtures.
 */
protected void setUp()
{
    empty               = new SetSort();
    one                 = empty.insert(new Integer(1));
    two                 = empty.insert(new Integer(2));
    three               = empty.insert(new Integer(3));
    oneAndTwo           = one.insert(new Integer(2));
    oneAndThree         = one.insert(new Integer(3));
    twoAndThree         = two.insert(new Integer(3));
    oneAndTwoAndThree   = oneAndTwo.insert(new Integer(3));

    bTrue = BoolSort.True();
    bFalse = BoolSort.False();
}

/**
 * @return a single Test that runs all the tests in this class
 */
public static Test suite()
{
    return new TestSuite(SetSortTest.class);
}

// Test Methods
// ------------
public void testContains()
{
    assert(oneAndThree.contains(new Integer(1)));
    assert(oneAndThree.contains(new Integer(3)));
    assert(! oneAndThree.contains(new Integer(2)));
    assert(! empty.contains(new Integer(1)));
}

public void testIsSubset()
{
    assert(empty.isSubset(one));
    assert(one.isSubset(oneAndTwo));
    assert(oneAndTwo.isSubset(oneAndTwoAndThree));

    assert(! one.isSubset(empty));
    assert(! oneAndTwo.isSubset(one));
    assert(! oneAndTwoAndThree.isSubset(oneAndTwo));

    assert(! oneAndTwo.isSubset(oneAndTwo));
    assert(! empty.isSubset(empty));
}

public void testIsSupset()
{
    assert(one.isSupset(empty));
    assert(oneAndTwo.isSupset(one));
    assert(oneAndTwoAndThree.isSupset(oneAndTwo));

    assert(! empty.isSupset(one));
    assert(! one.isSupset(oneAndTwo));
    assert(! oneAndTwo.isSupset(oneAndTwoAndThree));

    assert(! oneAndTwo.isSupset(oneAndTwo));
    assert(! empty.isSupset(empty));
}

public void testIsSubsetEq()
{
    assert(empty.isSubsetEq(one));
```

```
        assert(one.isSubsetEq(oneAndTwo));
        assert(oneAndTwo.isSubsetEq(oneAndTwoAndThree));

        assert(! one.isSubsetEq(empty));
        assert(! oneAndTwo.isSubsetEq(one));
        assert(! oneAndTwoAndThree.isSubsetEq(oneAndTwo));

        assert(oneAndTwo.isSubsetEq(oneAndTwo));
        assert(empty.isSubsetEq(empty));
    }

    public void testIsSupsetEq()
    {
        assert(one.isSupsetEq(empty));
        assert(oneAndTwo.isSupsetEq(one));
        assert(oneAndTwoAndThree.isSupsetEq(oneAndTwo));

        assert(! empty.isSupsetEq(one));
        assert(! one.isSupsetEq(oneAndTwo));
        assert(! oneAndTwo.isSupsetEq(oneAndTwoAndThree));

        assert(oneAndTwo.isSupsetEq(oneAndTwo));
        assert(empty.isSupsetEq(empty));
    }

    public void testSize()
    {
        assertEquals(0, empty.size());
        assertEquals(1, one.size());
        assertEquals(2, oneAndTwo.size());
    }

    public void testEquals()
    {
        assert(oneAndTwo.equals(oneAndTwo));

        SetSort oneAndTwo2 = new SetSort();
        oneAndTwo2 = oneAndTwo2.insert(new Integer(1));
        oneAndTwo2 = oneAndTwo2.insert(new Integer(2));
        assert(oneAndTwo.equals(oneAndTwo2));

        assert(oneAndTwo.equals(oneAndTwoAndThree.delete(new Integer(3))));

        assert(! one.equals(two));
        assert(! oneAndTwo.equals(oneAndThree));

        assert(! one.equals(new Integer(1)));
    }

    public void testNotEquals()
    {
        assertEquals(bTrue, SetSort.notEquals(oneAndTwo, oneAndThree));
        assertEquals(bFalse, SetSort.notEquals(oneAndTwo, oneAndTwo));
    }

    public void testIfThenElse()
    {
        assertEquals(oneAndTwo, SetSort.ifThenElse(bTrue, oneAndTwo, oneAndThree));
        assertEquals(oneAndThree, SetSort.ifThenElse(bFalse, oneAndTwo, oneAndThree));
    }

    public void testInsert()
    {
        assertEquals(oneAndTwo, one.insert(new Integer(2)));
        assertEquals("Immutability", empty.insert(new Integer(1)), one);
        assertEquals(oneAndTwoAndThree, oneAndTwo.insert(new Integer(3)));
        assertEquals("Immutabiliity", one.insert(new Integer(2)), oneAndTwo);
```

```
    }

    public void testDelete()
    {
        assertEquals(one, oneAndTwo.delete(new Integer(2)));
        assertEquals("Immutability", one.insert(new Integer(2)), oneAndTwo);
        assertEquals(oneAndTwo, oneAndTwoAndThree.delete(new Integer(3)));
        assertEquals("Immutability", oneAndTwoAndThree, oneAndTwo.insert(new Integer(3)));
        assertEquals(empty, empty.delete(new Integer(2)));
    }

    public void testUnion()
    {
        assertEquals(empty, empty.union(empty));
        assertEquals(one, empty.union(one));
        assertEquals(one, one.union(empty));
        assertEquals(oneAndTwo, one.union(two));
        assertEquals(oneAndTwo, two.union(one));
        assertEquals(oneAndTwoAndThree, oneAndTwo.union(three));
        assertEquals(oneAndTwoAndThree, three.union(oneAndTwo));
    }

    public void testIntersection()
    {
        assertEquals(one, oneAndTwo.intersection(oneAndThree));
        assertEquals(one, oneAndThree.intersection(oneAndTwo));
        assertEquals(two, oneAndTwo.intersection(twoAndThree));
        assertEquals(two, twoAndThree.intersection(oneAndTwo));
        assertEquals(empty, empty.intersection(one));
        assertEquals(empty, one.intersection(empty));
    }

    public void testDifference()
    {
        assertEquals(one, oneAndTwo.difference(two));
        assertEquals(one, oneAndTwoAndThree.difference(twoAndThree));
        assertEquals(one, one.difference(empty));
        assertEquals(one, one.difference(two));
        assertEquals(empty, empty.difference(empty));
    }

    public void testHashCode()
    {
        assertEquals(oneAndTwo.hashCode(), one.insert(two).hashCode());
        assert(one.hashCode() != oneAndTwo.hashCode());
    }
}
```

C.2.5 IOA File

```
automaton Set01
  signature
    internal a1
  states
    s: Set[Int],
    i: Int,
    b: Bool
  transitions
    internal a1
      eff
        s := {};
        s := {3};
        s := insert(i, s);
```

```
s := delete(i, s);
b := i ∈ s;
s := s ∪ s;
s := s ∩ s;
s := s - s;
b := s ⊂ s;
b := s ⊆ s;
b := s ⊃ s;
b := s ⊇ s;
i := size(s);
b := s = s;
b := s ≠ s;
s := if b then s else s
```

C.2.6 Generated Java Code

```
package ioa.runtime;

import java.io.*; import ioa.runtime.adt.*;

public class Set01 extends ioa.runtime.Automaton {
  SetSort s_v1 =
    (SetSort)SetSort.construct(new
                               Parameterization(new Class[]
                                                  {ioa.runtime.adt.IntSort.class},
                                                  new Parameterization[]
                                                    {new
                                                       Parameterization()}));
  IntSort i_v2 = (IntSort)IntSort.construct(new Parameterization());
  BoolSort b_v3 = (BoolSort)BoolSort.construct(new Parameterization());
  public void a0()
    {
      s_v1 = ((SetSort)SetSort.empty());
      s_v1 = ((SetSort)SetSort.singleton(((IntSort)IntSort.lit(3))));
      s_v1 = ((SetSort)SetSort.insert(i_v2, s_v1));
      s_v1 = ((SetSort)SetSort.delete(i_v2, s_v1));
      b_v3 = ((BoolSort)SetSort.isIn(i_v2, s_v1));
      s_v1 = ((SetSort)SetSort.union(s_v1, s_v1));
      s_v1 = ((SetSort)SetSort.intersection(s_v1, s_v1));
      s_v1 = ((SetSort)SetSort.difference(s_v1, s_v1));
      b_v3 = ((BoolSort)SetSort.isSubset(s_v1, s_v1));
      b_v3 = ((BoolSort)SetSort.isSubsetEq(s_v1, s_v1));
      b_v3 = ((BoolSort)SetSort.isSupset(s_v1, s_v1));
      b_v3 = ((BoolSort)SetSort.isSupsetEq(s_v1, s_v1));
      i_v2 = ((IntSort)SetSort.size(s_v1));
      b_v3 = ((BoolSort)SetSort.equals(s_v1, s_v1));
      b_v3 = ((BoolSort)SetSort.notEquals(s_v1, s_v1));
      s_v1 = (b_v3).booleanValue() ? s_v1 : s_v1;

  }

  public static void main(String[] args) {
ioa.runtime.Automaton.main(new String[] {"Set01"});
}

}
```

D Bibliography

References

[1] Eric Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns.*

[2] IOA Homepage. `http://theory.lcs.mit.edu/tds/ioa.html`

[3] Stephen J. Garland and Nancy E. Lynch. *The IOA Language and Toolset: Support for Designing, Analyzing, and Building Distributed Systems.* `http://theory.lcs.mit.edu/tds/papers/Lynch/IOA-TR.html`

[4] IOA Manual. `http://theory.lcs.mit.edu/tds/papers/Garland/ioaManual.ps.gz`

[5] John V. Guttag and James J. Horning, editors. *Larch: Languages and Tools for Formal Specification.* `http://www.sds.lcs.mit.edu/spd/larch/pub/larchBook.ps`

[6] IOA Code Generator. `http://theory.lcs.mit.edu/~josh/codegen.html`

[7] JUnit Testing Framework. `http://www.junit.org`

[8] Sun's Javadocs for `hashCode()`. `http://java.sun.com/products/jdk/1.1/docs/api/java.lang.Object.html`

[9] Anna E. Chefter. *A Simulator for the IOA Language.* `http://theory.lcs.mit.edu/tds/papers/Chefter/thesis.html`

[10] J. Antonio Ramirez-Robredo. *Paired Simulation of I/O Automata.* `http://theory.lcs.mit.edu/~ramirez/thesis/`

[11] Laura G. Dean. *FIXME: thesis title*

[12] Toh Ne Win. *Logging in the IOA Toolkit.* `http://theory.lcs.mit.edu/~tohn/logging.html`

[13] Toh Ne Win. *Dynamic Sort Notes.* `http://theory.lcs.mit.edu/~tohn/dynamic-sorts.html`