SML2Java: a source to source translator *

Justin Koser
jmk63@cornell.edu

Haakon Larsen
hl272@cornell.edu

Jeff Vaughan
jav28@cornell.edu

May 22, 2003

1 Introduction

SML2Java is a source-to-source translator from Standard ML (SML), a statically typed functional language, to Java, an object-oriented imperative language. A successful translator must emulate distinctive features of one language in the other. For instance, SML's first-class functions are mapped to Java's first-class objects, and many other functional features are translated to take advantage of Java's object-oriented style.

2 Goals

SML2Java was inspired by problems encountered teaching functional programming to students familiar with the imperative, object-oriented paradigm. It was developed for possible use as a teaching tool for Cornell's CS 312, a course in functional programming and data structures. To be a successful educational tool it is important that the translated code is as intuitive as possible within the imperative language paradigm.

On a broader level, we wish to show how functional concepts can be mapped to object-oriented imperative concepts through a thorough understanding of each model. In this regard, it becomes important not to force functional concepts upon an imperative language, but rather to translate these functional concepts to their imperative equivalents.

3 Translation

In this section, we will present and discuss the choices we made in our translation of SML into Java. Where it is pertinent, we will also discuss the relative benefits/drawbacks of our specific choice relative to other possible translations.

*This project was completed as a CS 490 (independent research) at Cornell University under the supervision of Dexter Kozen (kozen@cs.cornell.edu). The authors wish to express their gratitude for his helpful advice and guidance.

3.1 Primitives

SML primitives (i.e. integers, real numbers, strings, characters) are translated to wrapper classes *Integer2*, *String2*, etc. The wrapper classes merely wrap primitive operations on the translated primitives. Ideally one would want the primitives to be translated to their Java equivalents (i.e. java.lang.Integer etc), but unfortunately these do not support the required functionality (e.g. arithmetic operations), and hence we must implement our own.

In future revisions, it is our intent that SML primitives will be translated into their Java equivalents. In order to perform primitive operations on *Integer* etc. one will call wrapper functions of the form *Integer2.add*. That is to say, a function like *Integer2.add* will be the imperative equivalent to the SML function *Int.+* given the Java wrapper function *Integer*: it will take two *Integer*s as its parameters, and perform a *return new Integer(a.intValue() + b.intValue());*. This translation will greatly improve the ease by which the generated Java code can be merged with existing Java code.

3.2 Tuples and Records

As it turns out, the SML compiler compiles tuples down to records, which simplifies our job a bit. Thus, every tuple of the form *(exp1, exp2, ...)* becomes a record of the form {*1=exp1, 2=exp2, ...*}. This should not surprise the avid SML fan as SML will, even at the top level, represent a record of the form {*1="hi", 2="bye"*} as a tuple *("hi","bye"): string*string*.

A *Record* class represents the SML concept of a record in SML2Java. Every SML record then becomes an instance of this class in the translated Java code. The *Record* class contains a private data member, *myMapping*, of type java.util.HashMap. The SML concept of a record is thus reduced to a mapping from fields (which are of type *String*) to the data that they carry (of type *Object*), which we believe is a very reasonable reduction. The *Record* class also contains a function *add*, which takes a *String* and an *Object* as its parameters and adds these to the mapping. A record of length n will therefore require n calls to the *add* function. A record projection in the current scheme is little more than a lookup in the given records HashMap.

In the code example below, the lines that contain the word 'Pattern' form the foundation of what is to be generalized pattern matching in future revisions of SML2Java. Pattern matching would be done at runtime (not at compile time, as in SML), and would consist in comparing the types of members in the given expression to the types in the pattern. In its current form, the only supported feature is to extract the data from the various fields a record contains.

```
val a = {name="John Doe", age=20}
val b = ("John Doe", 20)
```

```
public static final Record a = (Record)
  ((new Record())
    .add("name", new String2("John Doe")))
    .add("age", (new Integer2 (20)));

 public static final Record b = (Record)
  ((new Record())
    .add("1", new String2("John Doe")))
    .add("2", (new Integer2 (20)));
```

3.3 Datatypes

An SML datatype declaration creates a new type with one or more constructors. Each constructor may be treated as a function of zero or one arguments. SML2Java treats this model literally. An SML datatype, *dt*, with constructors *c1, c2, c3 ...* is translated to a class. This class, also named *dt*, has static methods *c1, c2, c3 ....* Each such method returns a new *dt*.

Thus, SML code invoking a constructor becomes a static method call in the translated code. It is important to note that this proccess is different from the translation of normal sml functions. The special handling of type construnctors underscores their distinct character and greatly enhances translated code readability.

```
datatype qux = FOO | BAR of int

val myVariable = FOO
val myOtherVar = BAR(42)
```

```
public class TopLevel {
  public static class qux extends Datatype {

    protected qux(String constructor){
      super(constructor);
    }

    protected qux(String constructor,Object data){
      super(constructor, data);
    }

    public static qux BAR(Object o){
      return new qux("BAR", o);
    }

    public static qux FOO(){
      return new qux("FOO");
    }

  }
  public static final qux myVariable = (qux)
    qux.FOO();

  public static final qux myOtherVar = (qux)
    qux.BAR((new Integer2 (42)));

}
```

3.4 Functions

A *Function* class in Java, where every SML function becomes an instance of this class, encapsulates the concept of SML functions in our translation model. The Java *Function* class has only one member function, *apply*, which takes an *Object* as its only parameter and returns an *Object*. The *Function* class encapsulation is necessitated by the fact that functions are treated as values in SML. As a byproduct of this scheme, function applications become very intuitive: any application is translated to *Function_Name.apply(argument)*.

At an early design stage, the authors considered translating each function to a named class and a single instantiation of that class. While this model provides named functions that can be passed to other functions and otherwise treated as data, it does not easily accommodate anonymous functions. A strong argument

for choosing the current model of function translation is that anonymous instantiations of the *Function* class provides a natural way to deal with anonymous functions.

We believe this is a sufficiently general approach that can handle all issues with respect to SML functions (including higher-order functions). In fact, every SML function declaration (i.e. named function) is translated (by the SML compiler) down to a recursive variable binding with an anonymous function. This means that our treatment of anonymous functions and named functions nearly mirror each other and thus lends itself to code readability.

```
val getFirst = fn(x:int, y:int) => x
val one = getFirst(1,2)
```

```
public static final Function getFirst = (Function)
  (new Function () {
    Object apply(final Object arg) {
      final Record rec = (Record) arg;
      RecordPattern pat = new RecordPattern();
      pat.add("1", new VariablePattern(new Integer2()));
      pat.add("2", new VariablePattern(new Integer2()));
      pat.match(rec);
      final Integer2 x = (Integer2) pat.get("1");
      final Integer2 y = (Integer2) pat.get("2");
      return (Integer2) (x);
    }
  });

public static final Integer2 one = (Integer2)
  (getFirst).apply(((
  (new Record())
    .add("1", (new Integer2 (1))))
    .add("2", (new Integer2 (2)))));
```

3.5 Let Expressions

A *Let* interface in Java encapsulates the SML concept of a let expression. The *Let* interface has no member functions. Every SML let expression then becomes an anonymous instantiation of the *Let* interface with one member function *in* which has no parameters, but returns whatever type is appropriate given the original SML expression. The *in* function is called immediately following the instantiation.

A slightly different approach would be to have the *Let* interface contain the function *in*, which would have no formal parameters, but would return an *Object*. The upside to this would be its consistency with respect to our function translations (i.e. the *apply* function), but a possible downside is excessive casting

to and fro the *Object* type, which can greatly reduce readability.

One might also attempt to separate the *Let* declaration from the call to its *in* function. If implemented in the most direct manner, such a model would, like the previous one, require that the *Let* interface contain an *in* function. It is conceivable that this scheme would improve code readability. However, as one often has several *Let* expressions in the same name-space in SML, this model (as presented) would likely suffer from shadowing issues.

```
val x =
  let
    val y = 1
    val z = 2
  in
    y+z
  end
```

```
public static final Integer2 x = (Integer2)
  (new Let() {
    Integer2 in() {
      final Integer2 y = (Integer2) (new Integer2 (1));
      final Integer2 z = (Integer2) (new Integer2 (2));
      return (Integer2) (Integer2.add()).apply(((
      (new Record())
        .add("1", (y)))
        .add("2", (z))));
    }
  }).in();
```

3.6 Module System

Our translation of SML's module system is very straight-forward. SML signatures are translated to abstract classes. SML structures are translated to classes that extend these abstract signature classes. A structure class obviously only extends a given signature class in Java if the original SML structure implements the given SML signature. Structure declarations that are not externally visible in SML (i.e. not included in the implemented signature) are made private data-members in the generated Java structure class.

```
signature INDEX_CARD = sig
  val name : string
  val age : int
end

structure IndexCard :> INDEX_CARD = struct
  val name = "Professor Michael Jordan"
  val age = 31
  val super_secret = "This secret cannot be visibe to the outside"
end
```

```
private static abstract class INDEX_CARD  {
  public static final String2 name = null;
  public static final Integer2 age = null;
}

public static class IndexCard extends INDEX_CARD {
  public static final String2 name = (String2)
    (new String2 ("Professor Michael Jordan"));

  public static final Integer2 age = (Integer2)
    (new Integer2 (31));

  private static final String2 super_secret = (String2)
    (new String2 ("This secret cannot be visibe to the outside"));
}
```

4 Implementation

4.1 Overview

Our primary task was to translate high-level SML syntax to high-level Java syntax. Since there are several available implementations of SML, we chose to use the front end of one such implementation, Standard ML of New Jersey (SML/NJ). We use the development flavor of the interpreter (`sml-full-cm`) and call functions exported by that flavor to parse and type-check input SML code. We then translate the abstract syntax tree (AST) generated by SML/NJ to our own internal Java syntax and output the Java code in source form.

5 The Future of SML2Java

The current version of SML2Java translates many core constructs of SML, including primitives values, datatypes, functions, recursion, signatures and structures.

If the authors continue their work on SML2Java, pattern matching would likely be the first to be added to the list of translated constructs. Pattern matching is a great strength of SML, and something that would be interesting to translate cleanly into Java.

Parametric polymorphism is another construct the authors would like SML2Java to handle. There are several reasons why this was not supported in the present version, but primarily it was because generics would presently have come at the cost of too many other important features. Furthermore, Java 1.5 (due out late 2003) will inherently support this feature, and the authors think that waiting for Java's version would produce much cleaner code. Waiting for Java's implementation of generic types would also allow the authors to explicitly bring out the differences between the two implementations (Java's and SML's), and hopefully illuminate some strengths/weaknesses in either.

There are many other constructs the authors would like to support, but that are not considered as crucial as the above two. Among these other SML constructs are exceptions, vectors, open declarations, mutual recursion and functors. The majority of these should be implementable without a great deal of difficulty, but they would all be very valuable additions to SML2Java.

On a practical point, it is also an aim of the authors to be able to support multiple SML input files. The authors expect it will be possible to take advantage of programs already written by third parties to parse a collection of SML files (e.g. the Compilation Manager included with SML/NJ and whose source code is freely distributed) and resolve dependencies in order to be able to produce an SML AST.

6 Conclusion

SML2Java has greatly expanded the authors' understanding of SML, Java, and programming constructs in general. Understanding the essence of these constructs, and how this essence most cleanly could manifest itself in another programming paradigm is what required the most thought and effort. A successful abstraction (where success is defined in terms of how general and how intuitive the abstraction is) of such a construct is also the most rewarding part of the research. On the whole, the authors believe that the current incarnation of SML2Java is successful.