SML2Java: A Source to Source Translator

Justin Koser, Haakon Larsen, and Jeffrey A. Vaughan

Cornell University

Abstract. Java code is unsafe in several respects. Explicit null references and object downcasting can cause unexpected runtime errors. Java also lacks powerful language features such as pattern matching and first-class functions. However, due to its widespread use, cross-platform compatibility, and comprehensive library support, Java remains a popular language.
This paper discusses SML2Java, a source to source translator. SML2Java operates on type checked SML code and, to the greatest extent possible, produces functionally equivalent Java source. SML2Java allows programmers to combine existing SML code and Java applications.
While direct translation of SML primitive types to Java primitive types is not possible, the Java class system provides a powerful framework for emulating SML value semantics. Function translations are based on a substantial similarity between Java's first-class objects and SML's first-class functions.

1 Introduction

SML2Java is a source-to-source translator from Standard ML (SML), a statically typed functional language [8], 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 an SML let expression could conceivably be translated to a Java interface containing an 'in' function, where every let expression in SML would produce an anonymous instantiation of the let interface in Java. Similarly, many other functional features of SML are translated to take advantage of Java's object-oriented style. Because functional features such as higher-order functions must ultimately be implemented using first-class constructs, we believe one can only achieve a clean design by taking advantage of the strengths of the target language.

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. For the translator to be a successful educational tool, the translated code must be intuitive for a student with Java experience.

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.

2 Translation

This section discusses the choices we made in our translation of SML to Java. Where pertinent, we will also discuss the benefits and drawbacks of our design decisions.

2.1 Primitives

SML primitive types, such as *int* and *string*, are translated to Java classes. The foregoing become *Integer2* and *String2*, respectively. Ideally, SML primitives would translate to their built-in Java equivalents (e.g. *int* $\rightarrow$ *java.lang.Integer*), but these classes (e.g. *java.lang.Integer*) do not support operations such as integer addition or string concatenation [10]. We do not map directly to *int* and *string* because Java primitives are not derivatives of *Object*, cannot be used with standard Java collections, and are not compatible with our function and record translations. The latter will be shown in sections 2.2 and 2.4. Our classes, which are based on *Java.util.**, include necessary basic operators and fit well with function and record translation. While Hicks [6] addresses differences between the SML and Java type systems, he does not discuss interoperability. Blume [4] treats the related problem of translating C types to SML.

Figure 1 demonstrates a simple translation. The astute reader will notice several superfluous typecasts. Some translated expressions formally return *Object*. However, because SML code is typesafe, we can safely downcast the results of these expressions. The current version of SML2Java is overly conservative, and inserts some unnecessary casts. Additionaly, the add function looks quite complicated. This is consistent with other function translations which are discussed in section 2.4.

2.2 Tuples and Records

We follow the SML/NJ compiler (section 3) which compiles tuples down to records. 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/NJ will, at the top level, represent the record {*1="hi", 2="bye"*} and the tuple *("hi","bye"): string*string* identically.

The *Record* class represents SML records. Every SML record value maps to an instance of this class in the Java code. The *Record* class contains a private data member, *myMapping*, of type *java.util.HashMap*. SML records are translated to a mapping from fields (which are of type *String*) to the data that they carry (of type *Object*). The *Record* class also contains a function *add*, which takes a

Fig. 1. Simple variable binding

SML Code:

```
val x=40
val y=2
val z=x+y
```

Java Equivalent:

```
public class TopLevel {
  public static final Integer2 x = (Integer2)
    (new Integer2 (40));

  public static final Integer2 y = (Integer2)
    (new Integer2 (2));

  public static final Integer2 z = (Integer2)
    (Integer2.add()).apply(((
    (new Record())
      .add("1", (x)))
      .add("2", (y))));

}
```

String and an *Object* as its parameters and adds these to the mapping. A record of length n will therefore require n calls to *add*. Record projection is little more than a lookup in the record's *HashMap*.

2.3 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 instance of *dt*.

Thus, SML code invoking a constructor becomes a static method call in the translated code. It is important to note that this process is different from the translation of normal SML functions. The special handling of type constructors greatly enhances translated code readability.

A datatype translation is given in figure 3. As the SML langauge enforces type safety, constructor arguments can simply be type *Object*. Although more restrictive types could be specified, there is little benefit in the common case where the type is *Record*.

Fig. 2. Two records instantiated

SML Code:

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

Java Equivalent:

```
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)));
```

2.4 Functions

In our translation model, the *Function* class encapsulates the concept of an SML function. Every SML function becomes an instance of this class. The Java *Function* class has a single method, *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 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 the current model is that instantiating anonymous subclasses of *Function* provides a natural way to deal with anonymous functions.

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

Other authors have used different techinques for creating functions at runtime. For example, Kirby [7] uses the Java compiler to generate bytecode dynamically. While powerful and well suited for imperative programming, this approach is not compatible with the functional philosophy of SML.

In figure 4, the lines that contain the word "Pattern" form the foundation of what will, in future revisions of SML2Java, be fully generalized pattern matching. Pattern matching is done entirely at runtime, and consists of recursively com-

Fig. 3. A dataype declaration and instantiation

SML Code:

```
datatype qux = FOO | BAR of int

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

Java Equivalent:

```
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)));

}
```

paring components of an expression's value with a pattern. SML/NJ performs some optimizations of patterns at compile time [1]. However these optimizations are, in general, NP-hard [3] and SML2Java does not support them. Currently patterns are limited to records (including tuples), wildcards and integer constants.

Fig. 4. Named function declaration and application

SML Code:

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

Java Equivalent:

```
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)))));
```

2.5 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 becomes an anonymous instantiation of the *Let* interface with one member function, *in.* This function has no parameters and returns whatever type is appropriate given the original SML expression. The *in* function is called immediately following object instantiation.

A different approach would be to have the *Let* interface contain the function *in.* Here, *in* would have no formal parameters, and would return an *Object.*

The advantage to this would be its consistency with respect to our function translations (i.e. the *apply* function), but a possible disadvantage is excessive typecasting, 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. This scheme would improve code readability. However, as one often has several *Let* expressions in the same name-space in SML, this model would likely suffer from shadowing issues.

Fig. 5. Let expressions are translated like functions

SML Code:

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

Java Equivalent:

```
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();
```

2.6 Module System

Our translation of SML's module system is straightforward. SML signatures are translated to abstract classes. SML structures are translated to classes that extend these abstract signature classes. A structure class only extends a given signature class if the original SML structure implements the 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. This is demonstrated in figure 6.

Fig. 6. Translation of a signature and a structure

SML Code:

```
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 visible to the outside"
end
```

Java Equivalent:

```
public class TopLevel {
  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 visible to the outside"));

  }

}
```

3 Implementation

Our primary task was to translate high-level SML source code to high-level Java source code. As there are several available implementations of SML, we chose to use the front end of one, Standard ML of New Jersey (SML/NJ) [9]. We use the development flavor of the compiler (`sml-full-cm`) to parse and type-check input

SML code. We then translate the abstract syntax tree generated by SML/NJ to our own internal Java syntax tree and output the Java code in source form.

Taking advantage of the SML/NJ type checker gives us a strong guarantee regarding the safety of the code we are translating. To cite Dr. Andrew Appel, a program produced from this code ”cannot corrupt the runtime system so that further execution of the program is not faithful to the language semantics” [2]. In other words, such a program cannot dump core, access private fields, or mistake types for one another. It would be interesting to investigate whether these facts, combined with the translation semantics of SML2Java, imply that similar guarantees hold in the generated Java code.

Other properties of the Core subset of SML are discussed by VanIngwegen [12]. Using HOL [5], she is able to prove, among other things, determinism of evaluation.

4 Conclusion and Future Goals

The current version of SML2Java translates many core constructs of SML, including primitive values, datatypes, anonymous and recursive functions, signatures and structures. SML2Java succeeds in translating SML to Java code, while respecting the functional paradigm.

Parametric polymorphism is a key construct that the authors would like to implement in SML2Java. Java 1.5 (due out late 2003) will directly support generics [11], and we believe waiting for Sun’s implementation will facilate generating clean Java code. In addition, Java’s generics will resemble C++ templates, and our treatment of parametric polymorphism should highlight the relative merits of each approach.

We would like to add support for several less critical SML constructs. Among these are exceptions, vectors, open declarations, mutual recursion, functors, and projects containing multiple files. The majority of these should be implementable without excessive difficulty, and each is expected to be a valuable addition to SML2Java.

5 Acknowledgements

This project was performed as independant research under the guidance of Dexter Kozen, Cornell University. We would like to thank Professor Kozen for many insightful discussions and much valuable advice. We would also like to thank the following for helpful advice: Andrew Myers, Cornell University, and Tore Larsen, Tromsø University.

References

1. Aitken, William. *SML/NJ Match Compiler Notes* http://www.smlnj.org/compiler-notes/matchcomp.ps (1992)

2. Appel, Andrew W. *A critique of Standard ML* http://ncstrl.cs.princeton.edu/expand.php?id=TR-364-92 (1992)
3. Baudinet, Marianne and MacQueen, David. *Tree Pattern Matching for ML (extended abstract)* http://www.smlnj.org/compiler-notes/85-note-baudinet.ps (1985)
4. Blume, Matthias. *No-Longer-Foreign: Teaching an ML compiler to speak C "natively"* Electronic Notes in Theoretical Computer Science 59 No. 1 (2001)
5. Gordon, Melham *Introduction to HOL. A theroem proving environment for higher order logic* Cambridge University Press, 1993
6. Hicks, Michael. *Types and Intermdiate Representations* University of Pennsylvania (1998).
7. Kirby, Graham, et al. *Linguistic Reflection in Java* Software - Practice & Experience 28, 10 (1998).
8. Milner, Robin, et al. *The Definition of Standard ML - Revised.* Cumberland, RI: MIT Press, 1997.
9. SML/NJ Fellowship, The. *Standard ML of New Jersey* http://www.smlnj.org (July 29, 2003).
10. Sun Microsystems. *Java 2 Platform, Standard Edition, v 1.4.2 API Specification* http://java.sun.com/j2se/1.4.2/docs/api/ (July 18, 2003).
11. Sun Microsystems. *JSR 14 Add Generic Types To The Java Programming Language* http://www.jcp.org/en/jsr/detail?id=14 (July 24, 2003).
12. VanIngwegen, Myra. *Towards Type Preservation for Core SML* http://www.myra-simon.com/myra/papers/JAR.ps.gz