sml2java

a source to source translator

Justin Koser, Haakon Larsen,
Jeffrey Vaughan

PLI 2003

DP-COOL

CORNELL

Contents

- Overview of SML
- Overview of Java
- Motivation behind sml2java
- A Look at Translations
  - Key Ideas
  - Examples
- Conclusion

What We Like About SML

- SML has a powerful type system
  - Strong types prevent errors due to casting
  - Static typing prevents run-time type errors
- Pattern matching on data structures produces clean and intuitive code
- Parametric polymorphism allows generic functions while maintaining type safety

More Reasons We Like SML

- SML functions are powerful
  - Higher order functions facilitate writing compact and expressive code
  - SML compliers unwrap tail recursive functions
- Garbage collection makes references easy

What’s so Great About Java?

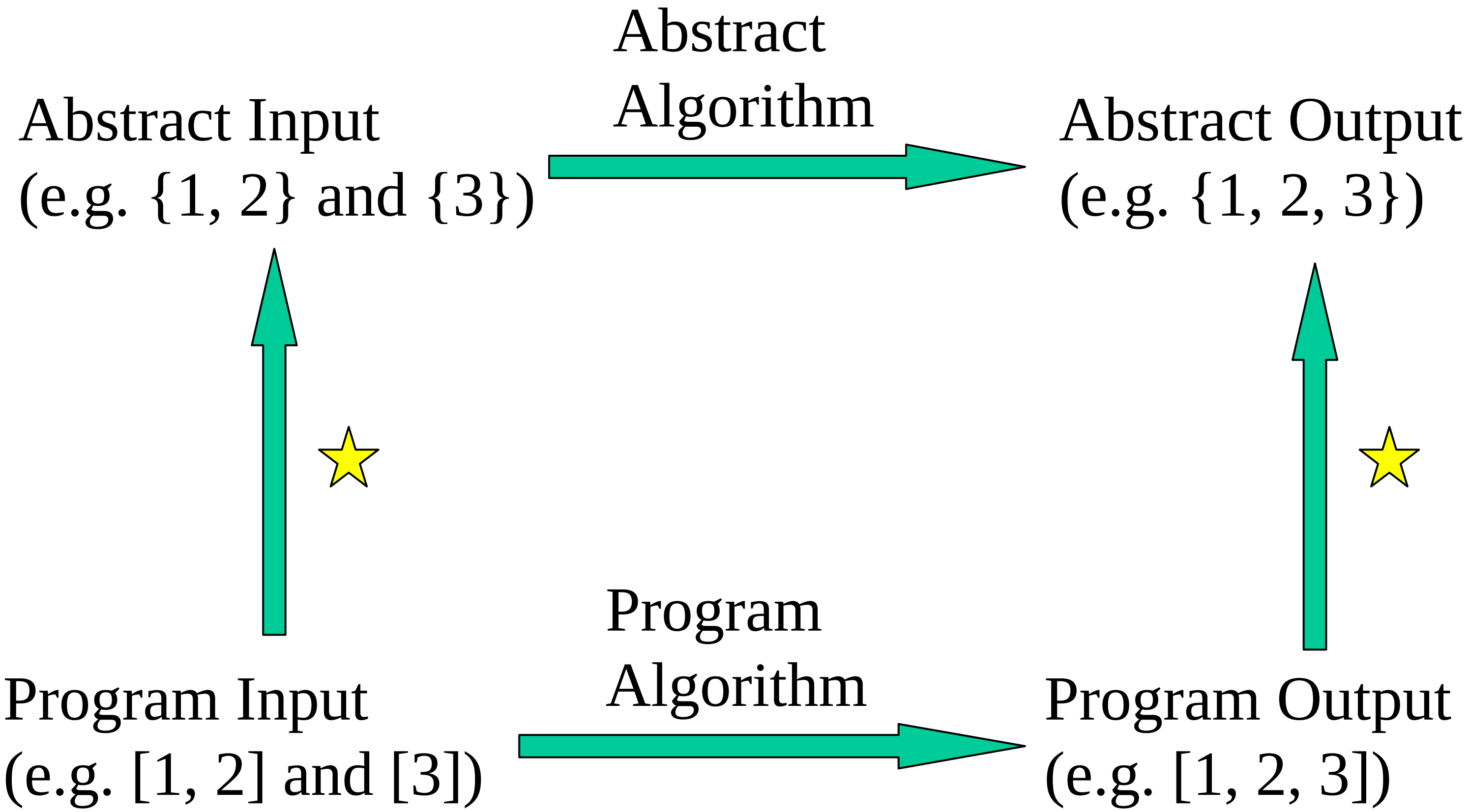
- Java is widely known and used in both industry and academia
- Java permits the programmer to write platform independent software
- Java’s first class objects can be built at run-time, and named or left anonymous
- Garbage collection makes references easy

Why sml2java ??

- Concepts underlying the translation could prove educationally valuable in teaching the functional paradigm
- Using a restricted subset of Java and a proof of correctness of the sml2java translator, the generated code would possess the same well-defined properties as the original SML

Abstraction Function

Example: union



Abstract Input
(e.g. {1, 2} and {3})

Abstract Algorithm

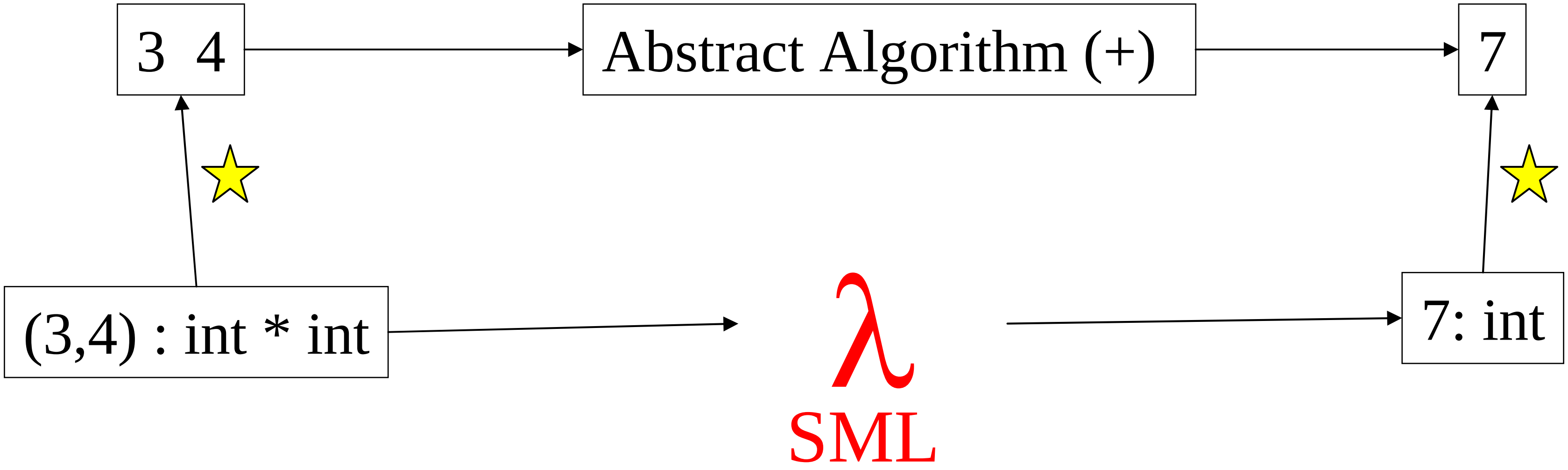
Abstract Output
(e.g. {1, 2, 3})

Program Input
(e.g. [1, 2] and [3])

Program Algorithm

Program Output
(e.g. [1, 2, 3])

Translation Diagram



3 4

Abstract Algorithm (+)

7

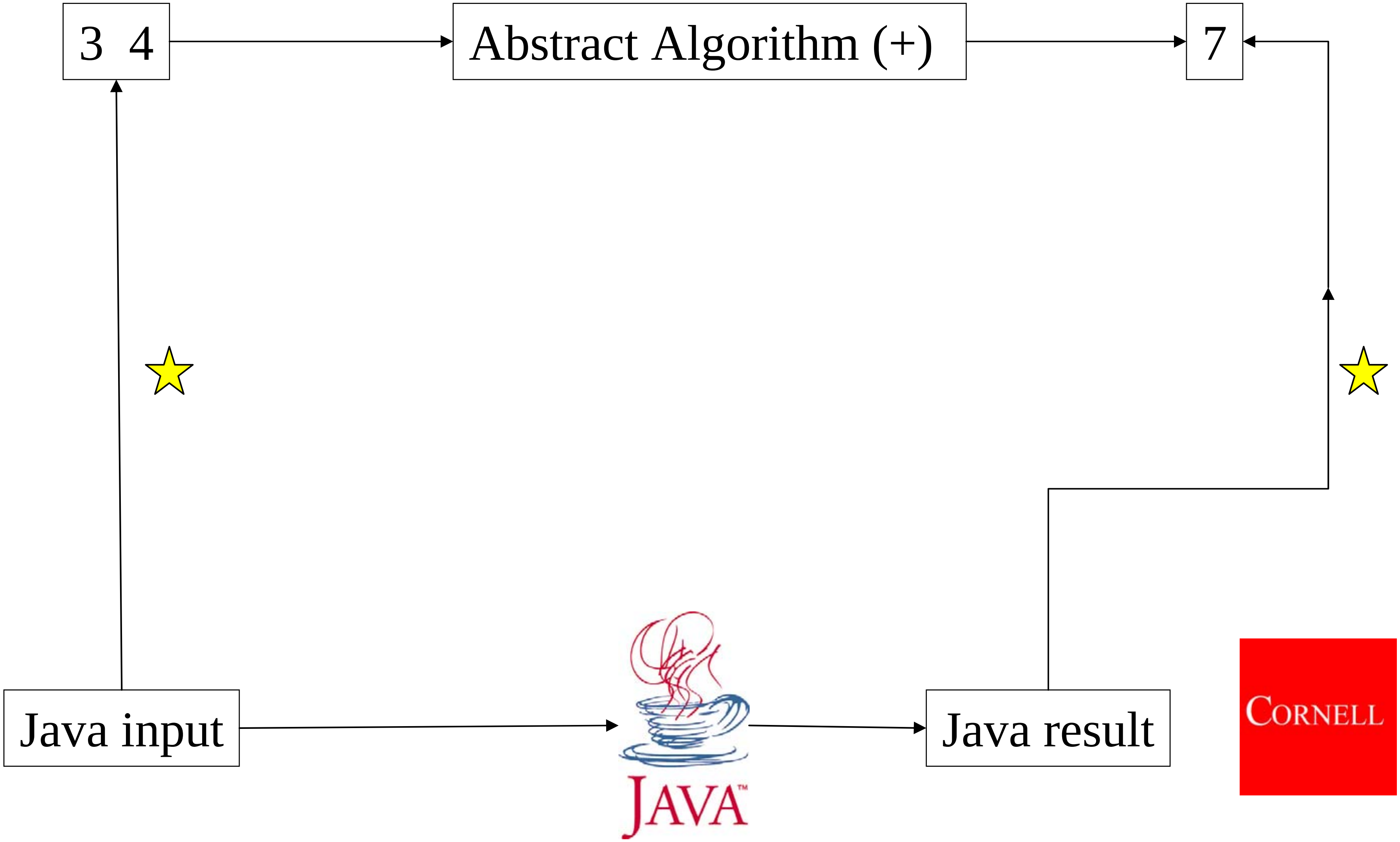
(3,4) : int * int

λ
SML

7: int

CORNELL

Translation Diagram



3 4

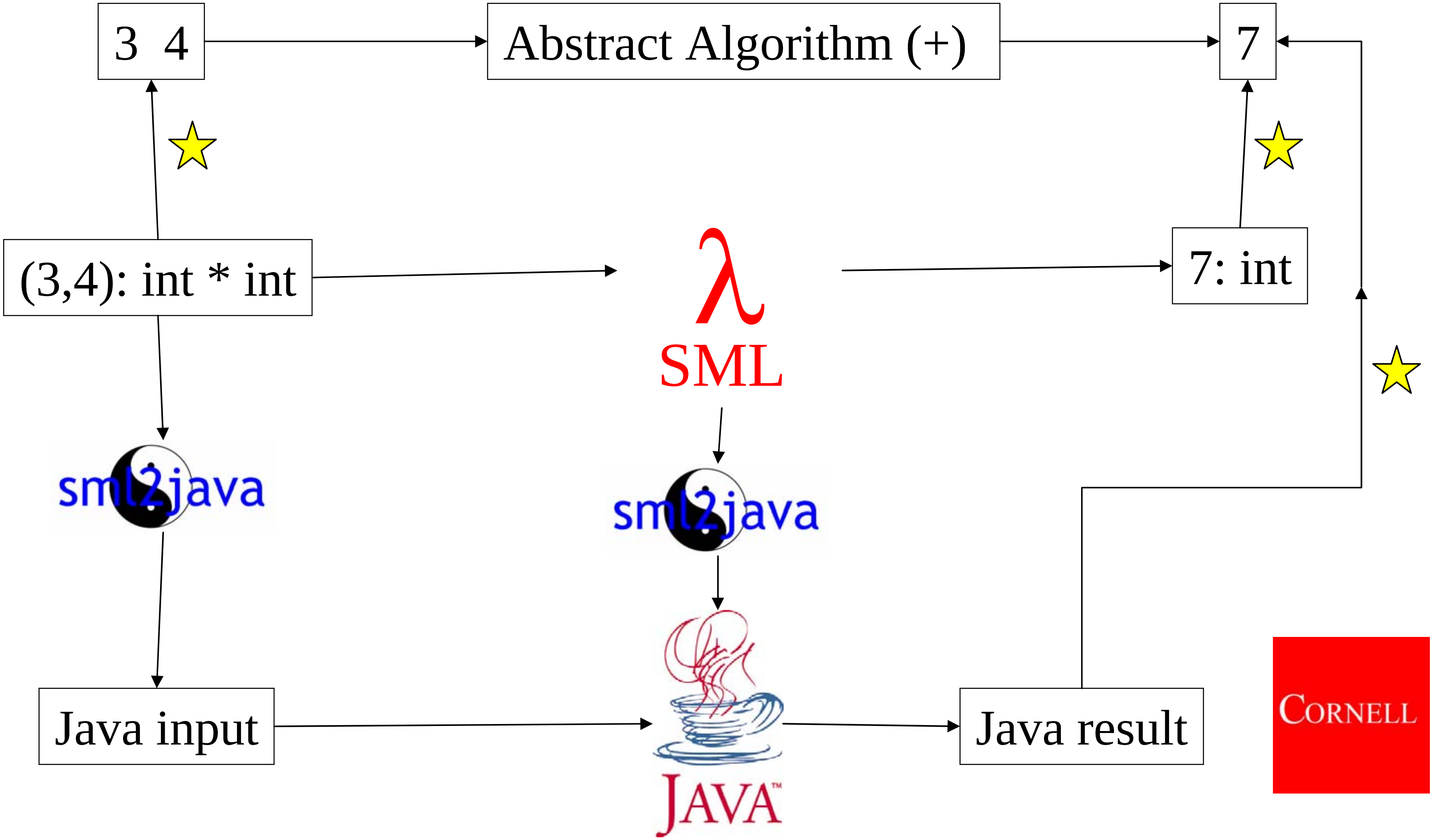
Abstract Algorithm (+)

7

Java input

Java result

Translation Diagram



3 4

Abstract Algorithm (+)

7

(3,4): int * int

7: int

Java input

Java result

CORNELL

Primitives

- SML primitives are translated into Java objects
- Java primitives (e.g. int, float) cannot be chosen as they would require translated functions to special-case for them
- An included library provides basic operations on the translated objects (e.g. add)

Tuples and Records

- SML tuples and records map unique field names to the values they contain
- Field names are set at compile time

- Java's HashMap maps unique keys to associated values
- A HashMap permits keys to be added at runtime

Thus a record of length n will require
n sequential additions to the HashMap

Datatypes

- SML datatypes create a new type with one or more constructors
- A datatype named dt with constructors c1, c2…c*n* produces a Java class named dt with static methods c1, c2…c*n*, which return an object of type dt
- Thus, SML code invoking a datatype constructor becomes a static method call in the translated Java code

Datatype Example

```
datatype qux = FOO of int
val myvar = FOO (42)
```

```
public static class qux extends
    Datatype {
  public static qux FOO
                         (Object o) {
    return new qux (“FOO", o);
  }
}
public static qux myvar =
  qux.FOO(new Integer (42));
```

Function Translations

- SML’s first class functions can be built at run-time, named or left anonymous, and passed to and returned from functions

- Java’s first class objects can be built at run-time, named or left anonymous, and passed to and returned from functions

Therefore,

(SML → Java) → (functions → objects)

CORNELL

Functions

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

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

Let Expressions

- SML Let expressions allow for N>1 variable bindings (where binding i can use bindings 1…i), which then can be used in a single expression, which is the result of the whole expression

- A Java function inside a class allows for N>0 variable bindings (where binding i can use bindings 1…i), which can then be used in a return expression, which is the result of the entire function

Let Expressions

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

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

Let Expression Options

- A Let [Java] interface with one function, in, with no parameters and returning Object
- A Let [Java] interface with no functions, where every instance would contain an in function that returns an appropriate type
- Separate the Let clause from the in clause

Module System

- SML signatures cannot be instantiated
- They declare variables that structures implementing these signatures must implement

- Java abstract classes cannot be instantiated
- They declare functions that non-abstract classes extending an abstract class must implement

Module System Example

```
signature ID = sig
  val name : string
end

structure Id :> ID = struct
  val name = “1337 h4x0r"
  val secret = “CIA supports ..."
end
```

```
private static abstract class ID {
  public static String name = null;
}

public static class Id extends ID {
  public static String name =
    (new String (“1337 h4x0r"));
  private static String secret =
    (new String(“CIA supports ..."));
}
```

Conclusion

- One can successfully translate many core constructs of SML elegantly into Java
- Some interesting constructs (e.g. parameterized polymorphism) remain
- While the ideas behind the translation have educational value, the implementation does not
- Investigating whether a “proof of correctness” (i.e. to ensure the safeness of translated code) is possible

CORNELL

Acknowledgements

- We would like to thank Professor David Gries and Ms. Lesley Yorke for securing financing for our presentation
- We would like to thank Professor Dexter Kozen for his invaluable assistance and guidance