Automatic Construction of Java Programs from Functional Program Specifications

Md. Humayun Kabir
Dept. of Computer Science and Engineering
Jahangirnagar University
Savar, Dhaka-1342, Bangladesh

Abstract—**This paper presents a novel approach to construct Java programs automatically from the input functional program specifications on natural numbers from the constructive proofs of the input specifications using an inductive theorem prover called Poiti′n. The construction of a Java program from the input functional program specification involves two phases. The theorem prover is used to construct a higher order functional (HOF) program from the input specification expressed as an existential theorem. A set of mapping rules for a Programming Language Translation System (PLTS) is defined for translating functional expressions to their semantic equivalent Java code. The generated functional program is translated into intermediate Java code in the form of a Java function using the PLTS module. The generated Java function requires a small refinement to obtain a syntactically correct Java function. This Java function is encapsulated within a user defined Java class as a member operation, which is invoked within a Java application class consisting of a *main* function by creating objects resulting in an executable Java program. The constructed functional program and the generated Java program both are correct with respect to the input specification as they produce the same output.**

Keywords—Functional Program Specification; Existential Theorems; Higher Order Functional Program; Mapping Rules; Programming Language Translation System; Java Program; Refinement

I. Introduction

Automatic construction of executable programs from the input program specifications is really a difficult task. A number of theorem provers are available, for example, Poiti′n, Nuprl1, and Coq, which can be used to construct functional programs from the proofs of their specifications [1,2,3,4,5,6]. Several code generation tools e.g., Rational Rose, Microgold and Umbrello have been developed for automatic generation of Java or C++ program code from UML design specification expressed in terms of class diagrams for a particular computing problem solution [1]. These tools can be used to generate architectural code when class details in UML notation, i.e., class name, attributes, operations and class relationships are provided within the class diagram. The details code for each class operation has to be provided by the programmer. The generated code can only be verified by executing the code to see whether it provides the desired output and functionality. The correctness, reliability and completeness of the generated programs fully depend on UML class design expertness and programming skill of the designer to encode the problem. The verification is done manually [1] by the designer to check its correctness.

Formal software development using mathematical rules aids automatic or semi-automatic program development from their specifications using their correctness proofs. Automatic construction of higher order functional programs from the proofs of their specifications using metasystem transition proofs has been developed [2,3,1]. In the synthesis of functional programs from specifications, various approaches exist in which either a program is extracted from the proof of the specification [4,6], or transformation rules are applied to the specification to obtain a program [7].

Poiti′n [2,3,8] is an inductive theorem prover, which can be used to perform constructive proof of an existential theorem expressed in a simple higher order functional language (HOFL) to extract functional program from the proof of a non-executable input specification [2]. The constructed program is an executable functional program in the source language (SL). The language of Poiti′n is untyped and non-strict with first-order quantifiers. In this paper, input specifications on natural numbers are considered for program construction. The universal variables are intended to be used as input variables are not quantified, and therefore must remain within the constructed HOFL program. The existential variables are ANY quantified with explicitly defining their data types (e.g. *nat* for natural number). These are the witness variables which construct the output value. All of these variables are natural number variables. The existential theorem with the required function definitions, which is used as the input program specification, describes the properties of the desired program to be constructed [1]. The theorem prover applies distillation program transformation algorithm [9,8,10,2,3] to the input specification to obtain a distilled program, and applies the proof rules to this program to verify the correctness of the input specification. A set of program construction rules is applied to the distilled program [2,3] to construct a functional program if the specification is proved correct.

Java is an attractive platform independent object-oriented programming language to the object-oriented software development community. So far we know from on-line literature search, no research work is found on automatic construction of Java programs from input specifications, and

This research work is based on the work presented in [1] which was funded by Jahangirnagar University, Savar, Dhaka, Bangladesh under the research grants of 2009-2010 in the Faculty of Mathematical and Physical Sciences.

[1]Nuprl System:http://www.nuprl.org/html/NuprlSystem.html

the available theorem provers can only be used to construct functional programs [1,2,3,4,5]. This paper presents a new approach for the construction of Java programs from the input functional program specifications expressed in the functional language of the theorem prover Poiti′n. A PLTS module applies a set of mapping rules to translate the constructed HOFL program into an equivalent Java function which is further refined to obtain a correct Java function. An executable Java program is developed to invoke this Java function which computes values similar to that of the HOFL program.

The rest of the paper is organized as follows. Section II presents the language of the theorem prover Poiti′n. Section III provides an overview about the programming language translation system (PLTS), and the related work. Section IV presents the system architecture for the automatic construction of Java programs from input functional program specification. Section V describes the relevance of the proposed system architecture for Java program construction from input functional program specification. Section VI gives an overview of higher order functional program construction from input functional program specification. Section VII defines a set of rules for translating the higher order functional program expressed in the language of Poiti′n to Java code with the refinement steps. Section VIII describes the process of constructing executable Java program using the generated Java code by defining Java classes with the required refinement. Section IX describes the implementation and results, and finally, section X concludes with a guideline to the future work.

II. Language

The language of the theorem prover Poiti′n is defined as a simple higher order functional language. A finite set of free variables {*u, v, x, y, z, u', v'*, …} with any number of renaming of these variables, a finite set of list variables {*us, vs, xs, ys, zs, us', vs'*, …} with any number of renaming of these variables, and a finite set of function symbols {*f*, *f*0, *f*1, g, *h*} are considered. The notation e_i (for $i = 1$ to n) is used to represent any expression in the language. A simple expression in the language can be a variable *x*, a constructor *c*, a constructor application $c\ e_1\ ...\ e_n$, a lambda expression $\lambda x.e$, a function variable *f*, or an application $e_0\ e_1$ [2,3,8,9]. The language also contains complex **case** and **letrec** expressions. A **case** expression is defined as **case** e_0 **of** $p_1 : e_1 \mid ... \mid p_k : e_k$ consisting of *k* alternate branches. The pattern p_i appearing in the i^{th} **case** branch is defined by the expression $c\ x_1\ ...\ x_n$ where *c* is a constructor and x_i are bound variables. A **letrec** expression is defined as **letrec** $f = e_0$ **in** e_1, where e_0 may contain a recursive call to the function *f* [2,3,8,9].The language has two first order quantifiers ALL and EX for quantifying universal and existential variables along with an ANY quantifier in order to specify the existential witness contained in the input program specification [2,3]. The input specification can be expressed in any of the following forms [2]:

ANY *y*:*datatype.e* (i)

ANY *y*:*datatype.pre* → *post* (ii)

where *y* is the existential variable representing existential witness to be computed, *datatype* is the type of the witnessing variable. In expression (i), *e* is the expression representing the properties of the program to be constructed consisting of functions and relations about natural numbers. Specification (ii) contains a pre-condition (*pre*), which is a constraint to restrict the program to be constructed from the proof of the specification to generate only the desired witness values. The input specification contains quantifier-free universal input variables, which must remain within the functional program constructed from the input specification [2,3]. The sub-expressions *pre* and *post* are valid expressions in the language.

A program, conjecture or program specification is expressed in the language in the following form [2,3,8,9]:

e
where
$f_1 = e_1$;
…
$f_n = e_n$;

Conjectures to be proved are defined in the form ALL $x_1…x_n$.EX $y_1…y_n.e$ where x_i and y_i are universally and existentially quantified variables respectively.

III. Programming Language Translation System (PLTS)

A programming language translation system (PLTS) can translate expressions in a source programming language into expressions in the target language (TL). For example, a C++ expression can be translated to a Java expression using a C++ to Java translator. The complexity of programming language translation depends on the syntactic and semantic gap between the source and target languages. Significant research works have been done for developing PLTSs for generating Java code from various source programming languages [1,11,12,13,14]. An approach to compile Standard ML program to Java bytecode has been presented in [11]. Some translation approaches have been proposed to obtain Java code from Haskell code [12], C code from ATLAS code [13], and Java code from COBOL code [14].

The author presents an approach to obtain Java program from a functional program specification in this paper. The proposed method translates the constructed functional program in the higher order functional language of Poiti′n to Java code using a set of mapping rules of the PLTS module. Poiti′n uses a non-strict and untyped higher order functional language, whereas Java is a strict and typed language. Because of a small number of instructions in the source language, we can obtain a few of the Java expressions for the input functional language to construct equivalent Java code. A prototype system has been developed which can be used to obtain Java code in the form of a Java function from a functional program [1] by translating the HOFL program constructed from the input functional program specification.

IV. System Architecture

The architecture of the proposed system for automatic Java program construction from functional program specification is represented in Fig. 1. In the proposed system, the inductive theorem prover Poiti'n constructs a functional program from the input specification. The parsing module extracts the source language constructs from the constructed functional program. The mapping module applies a set of translation rules to translate the functional program to Java code.

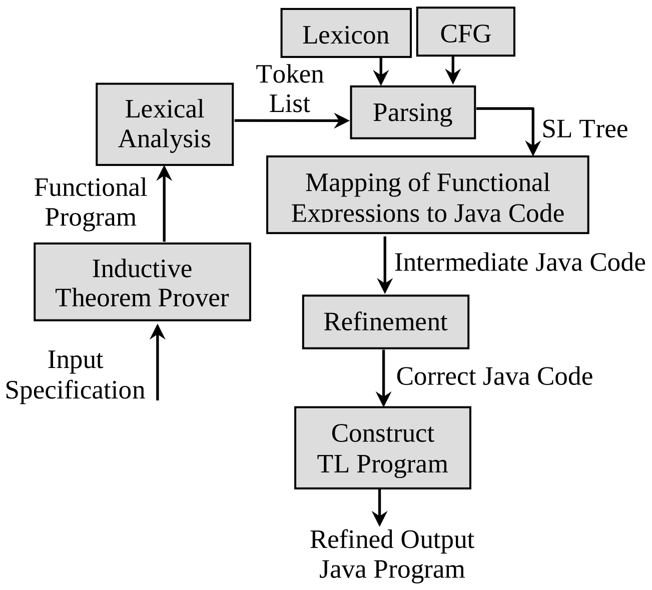

Fig. 1. System Architecture for Program Construction [1]

The process of Java program construction from a functional program specification involves several phases. At first, a higher order functional program is constructed from the input specification. In the next phase, the constructed functional program is translated to obtain equivalent Java code in the form of a Java function [1]. This Java function is further refined in the refinement phase to obtain a correct Java function computationally equivalent to that of the higher order functional program. In the construction phase, the Java function is encapsulated within a Java class as a member operation. This operation can be invoked by creating objects within a Java application class consisting of a *main* function to obtain an executable Java program.

V. RELEVANCE

Automatic construction of Java programs from input program specifications to solve a computational problem using program construction system is of great research interest to the software development research community. As far we know from the online literature, there is no work done so far to construct Java program automatically from input program specification expressed in Java language. The theorem prover Poiti'n can be used to construct functional programs from the input specifications expressed in its functional language.

A programming language translation system (PLTS) is proposed which can be used to translate simple higher order functional programs to Java functions. The research presented in this paper focuses on the construction of Java program from a functional program specification to solve a particular computational problem. Our intention is to automatically construct Java programs which can perform the same computations as that of the constructed functional programs. As there exists a functional program for the input functional program specification, which can be proved true by construction of that program from the constructive proof of the input specification using the theorem prover Poiti'n, there should exist a corresponding Java program to compute the same output as that of the constructed functional program. This paper presents an architecture shown in Fig. 1 for such Java program construction from input functional program specification about natural numbers only. The generated Java function can be used to develop an executable Java program. The proposed program construction system will lessen the burden of a programmer of writing details Java program code for those computational problems specified in HOFL of Poiti'n, which have their constructive proofs in Poiti'n to construct HOFL programs.

VI. FUNCTIONAL PROGRAM CONSTRUCTION

In the proposed system, the user has to define an input specification about natural numbers in the language of Poiti′n to automatically construct a functional program to solve a particular computational problem. The input specification describes the properties of the program to be constructed in terms of constraints and input/output relationship [2,3]. The input specification is expressed in the form of an existential theorem in terms of quantifiers, variables, type of the witnessing variable, predicates and functions. An equivalent higher order functional program is obtained from this specification using a set of program transformation rules called distillation [2,3,8,9]. The theorem prover applies a set of constructive proof rules [2,3,9] to this distilled program to construct a functional program for witness construction. The constructed functional program satisfies the properties described in the specification, and can be used to compute the value(s) of the existential witness which satisfies the program specification [1,2,3]. This is the actual purpose of the computational problem to be solved for which the input program specification was defined.

Consider the program specification defined by expression (1) as shown below.

```
ANY y:nat.or(eqnum(double y) x)
            (eqnum(Succ(double y)) x)          (1)
where
or = λx.λy.case x of
            True ⇒True
            | False ⇒ y
            | Bottom ⇒ y
eqnum  =λx.λy.case x of
                Zero ⇒(case y of
                        Zero ⇒ True
                       | Succ(y')⇒ False)
            | Succ(x')⇒(case y of
                        Zero⇒ False
                       | Succ(y')⇒eqnum x' y')
double = λx.case x of
            Zero ⇒Zero
           | Succ(x')⇒Succ(Succ(double x'))
```

The specification states that *the natural number y is to be constructed such that for all values of x, x is either double of y or the successor of the double of y*. The constructed output functional program given by the following expression (2) is obtained from the above input functional program specification (1).

```
Letrec f0 = λx.case x of                                  (2)
                Zero ⇒Zero
              | Succ(x')⇒case x' of
                              Zero ⇒Zero
                            | Succ(x")⇒Succ(f0 x")
in f0 x
```

The functions and relations used in specification (1) have their usual meaning and definitions using **case** expression [1,2]. In the definitions, the value *Bottom* represents an undefined value of a three-valued logic, i.e. *True*, *False*, *Bottom*. Poiti'n constructs a functional program defined by expression (2) from the input specification (1) [1,2]. Expression (2) can be redefined by expression (3) in the form of a HOFL program both computing the same output value for the same input. However, in this paper, expression (2) is used for the translation purpose.

```
f0 x                                                      (3)
where
f0 = λx.case x of
          Zero ⇒Zero
        | Succ(x')⇒case x' of
                        Zero ⇒Zero
                      | Succ(x")⇒Succ(f0 x");
```

Within the expressions, the symbol λ is used for variable binding and x' represents the predecessor of x, i.e., x-1. The variable *f0* is a recursive function which is defined by using a **letrec** expression. Within expression (1), the functions and relations used are on natural numbers, and the universal input variable x and the existential variable y under construction both are of type *nat*. As we have to input a natural number x to construct the witness y for it using the constructed HOFL program, x must remain within the HOFL program, i.e., x is quantifier-free universal variable. In evaluating the constructed HOFL program given by expression (2) using a natural number input for x, the argument x decreases by 2 in each recursive call to the recursive function *f0* till x reduces to 0 using the successive steps [1,2]. Verifying the program given by expression (2), we see that the program constructs a value of y for each input value of x satisfying the input specification (1).

VII. Translation of Functional Program to Java Code

The higher order functional program constructed by the theorem prover is usually expressed by using a **letrec** expression defining a recursive function, which is translated to a Java function by the PLTS module. A set of the mapping rules Ƭ is defined for the PLTS module as shown below [1] for translating the HOFL expressions to intermediate Java code.

```
Var →Ƭ<v> ϕ     = <int v> ϕ                               (Ƭ1)
VarRen→Ƭ<v'> ϕ = <v - 1> ϕ,      if v=v                  (Ƭ2)
                = <Ƭ<v> - 1> ϕ, Otherwise
                  (if v is a renaming of v)
VarList→Ƭ<vs> ϕ = <int vs[]> ϕ                            (Ƭ3)
Cons →Ƭ<Zero> ϕ = <0> ϕ                                   (Ƭ4)
ConsApp→Ƭ<Succ(e)> ϕ = < Ƭ<e> + 1> ϕ                     (Ƭ5)
FuncVar→Ƭ<f> ϕ = <f() {}> ϕ,  if f∉ϕ                     (Ƭ6)
FuncApp→Ƭ<f e1,..., en> ϕ = <f(Ƭ<e1>,...,Ƭ<en>);> ϕ      (Ƭ7)
CaseExpr→Ƭ<case x of                                      (Ƭ8)
              Zero: e1
            | Succ(x'): e2> ϕ
          = <switch (x)
              {case 0: Ƭ<e1>
                 break;
               default: {x' = x-1; Ƭ<e2>;} }> ϕ
FuncDef →Ƭ<f = λx1. ... .λxn.e>   ϕ                      (Ƭ9)
          = <public void f(int x1, ... , int xn)
              {Ƭ<e>}> ϕ
Letrec →Ƭ<letrec f = λx1.... .λxn.case x1 of             (Ƭ10)
                          Zero: e1
                        | Succ(x1'): e2
       in f x1 ... xn> ϕ
          = <public void f(int x1, ... , int xn)
              {Ƭ<case x1 of
                   Zero: e1
                 | Succ(x1'): e2>} ϕ ∪ {x1,...,xn}
                                      | f(x1,...,xn);>
```

The constructed functional program is tokenized to produce a token list which is input to the parsing module along with lexicon and the context free grammar (CFG) of the source language as shown in Fig. 1. The parsing process generates several component sub-expressions in the form of a tree by processing this token list. The mapping rules T are applied to the component sub-expressions to obtain their corresponding Java code. The generated Java code is not executable in its current form.

A HOFL expression can be defined by the following rule:

HOFLexpr→<Var> | <Varlist> | <Cons> | ConsApp | <CaseExpr> | <FuncApp> | <FuncVar> | <FuncDef> | <Letrec> | …

where '|' represents switching between different functional language constructs [1].

The general form of a mapping rule is defined as

HOFLexprType → T<*HOFLexpr*> <*JavaCode*>

where the variable *HOFLexprType* represents the *type* of the HOFL expression under translation, *HOFLexpr* is the HOFL expression to be translated, and *JavaCode* is the equivalent Java code of this HOFL expression.

Each of the primitive HOFL expressions has its corresponding equivalent Java code in its basic form where the source and target language constructs have same variable name. In these rules, f and $f0$ denote the function variable, x, x', x_1, x_1', y, y' and vs are data variables, and e, $e1$, $e2$ are expressions. The environment variable ϕ is used to store the universal input variables appearing within the input specification. The expression type, keywords, identifiers and sub-expressions of a HOFL expression are determined during parsing of the constructed functional program, which are input to the translation/mapping module for further processing of the functional expression [1].

Rule Ʈ1 encounters a variable v in HOFL syntax, and since the HOFL program contains only natural number variables as specified in the input program specification, it is translated to an integer type variable in Java. Rule Ʈ2 encounters a renaming v' of a natural number variable v. Since the renaming occurs only at the recursive steps and as v' is a sub-component of v, hence v is decremented to its predecessor by decrementing v by 1. If $\boldsymbol{v}$ is a renaming of v, then $\boldsymbol{v}$ is further translated using Ʈ. Rule Ʈ3 encounters list type variable vs of natural numbers in HOFL syntax, and it is translated to an integer array variable in Java syntax. Rule Ʈ4 and Rule Ʈ5 deal with constructors. Rule Ʈ4 encounters the constructor *Zero*, which is translated to an equivalent Java integer number 0. Rule Ʈ5 encounters the constructor application *Succ*(*e*). In this rule, 1 is added with the result of translating the argument *e*. Rule Ʈ6 translates a HOFL function variable *f* with no arguments to a Java function *f*(). Rule Ʈ7 encounters a function application of the function *f* with *n* number of arguments e_1 ... e_n. The PLTS translates this function application to a Java function call to the function *f* with the results of separately translating the arguments e_1,..., e_n as the function arguments. Rule Ʈ8 encounters a HOFL **case** expression which is translated to a *switch* statement in Java syntax, and the HOFL sub-expressions in the **case** branches are recursively translated to their equivalent Java code. Before translating the case branches, any renamed variable occurring within the **case** branches is searched within the environment variable ϕ, and it is checked to see whether it is a renaming of any of the variable found within ϕ. The renamed variable is initialized with decrementing the original **case** selector variable by 1 for each renaming. Rule Ʈ9 translates a HOFL function definition of *f* with *n* bound variables. The lambda (λ) bound variables x_1 ... x_n used in the body of the function *f* are local to the function *f*, which become the formal parameters *int* x_1, ... , *int* x_n of the corresponding Java function *f*. The body of the Java function *f* is obtained by translating the HOFL expression *e* of the function *f*. Rule T10 translates a HOFL letrec expression which defines a function *f* with *n* parameters including a function call to *f*. The λ bound variables x_1 ... x_n used in this expression are local to the function *f*, which become the formal parameters *int* x_1, ... , *int* x_n of the corresponding Java function *f*. The body of the Java function *f* is obtained by translating the **case** expression of the **letrec** expression. The function call *f*(x_1 ... x_n) used in the tail of the **letrec** expression is translated to a Java function call *f*(x_1, ..., x_n), and the variables x_1, ..., x_n are inserted into ϕ.

In the application of the rules Ʈ to an HOFL expression, the matching of any component expression contained in the constructed functional program with the appropriate mapping rule skeleton is performed on the skeleton of the HOFL component contained in the appropriate mapping rule [2,3,8,9].

Example

Consider the translation of a HOFL **letrec** expression which defines the function *f* as given by expression (4) into Java code using the rules Ʈ of the PLTS module. In this expression, *x* is a natural number variable which is decremented by 1 in each recursive call to the function *f* until *x* reduces to 0.

letrec f = λx.**case** x **of**
Zero : Zero (4)
| Succ(x'): f x'
in f x

The PLTS module generates the intermediate Java code as shown below which defines the Java function *f* using the mapping rules Ʈ. The Java function *f* needs to be refined to obtain a syntactically correct Java function.

// Intermediate Java Code in the form of a function definition:

```
public void f(int x)
{
  switch (x)
     {case 0:
          0;
        break;
       default:
         x' = x -1;
         f(x');
 }
 }
```

// Function call:

```
      f(x);
```

A. Refinement of the Java Code

The refinement phase makes few changes to the generated Java code of the function *f* as shown above resulting in the refined correct Java code as shown below.

```
public int f(int x)
  {
     switch (x)
        {
           case 0:
               res = 0;
               break;
        default:
               x = x -1;
               res = f(x);
        }
   return res;
  }
```

During refinement, at first, the *void* type of the function *f* is converted to *int* type. This change is mandatory as the constructed HOFL program defined in the form of a **letrec** function returns a natural number value as the output of the function, so the generated Java function obtained from the HOFL function must have the same return type declared in the function header or function prototype declaration, i.e., *int* type in Java. As it is difficult to handle the return type of the generated Java function within the rules Ʈ, the return type of the function is added during the refinement phase of the program construction system as shown in Fig. 1. Second, the statements, expressions or values which contribute to final result are identified, and an output variable, e.g. *res*, is initialized with these components. The output variable is declared as an attribute of a Java class in which this function will be encapsulated as a member operation. Third, a return statement is added to return the output from the Java function *f*. Finally, as the renamed variable, e.g., *x'* is not a valid identifier in Java, so, it is replaced with the value given in terms of the original variable *x*. The initialization of the original variable to its updated value can be defined in terms of the original variable itself in Java, e.g., *x* = *x* – 1.

VIII. Java Program Construction

In the construction phase, an executable Java program can be developed using the Java function obtained after refinement such that both of the HOFL program constructed from the input functional program specification and the developed Java program from this HOFL program perform the same computation.

Consider the generation of Java code using the rules ⁊ by translating the **letrec** expression (2) which is the HOFL program constructed from specification (1). The application of the rules ⁊ to expression (2) translates it to the intermediate Java code which defines function *f*0 is shown below.

```
public void f0(int x)
  {
   switch (x)
    {
     case 0:
            0;
        break;
     default:
     {
        x' = x -1;
        switch (x')
         {
           case 0:
                 0;
              break;
           default:
           { x'' = x -1-1;
              f0(x'') + 1;
           }
        }
      }
    }
  }
```

The function *f*0 can be used to compute the natural number *y* for an input *x* such that *x* is either double of *y* or the successor of the double of *y* as stated in the program specification (1). The developed Java code of function *f*0 is a bit difficult for the beginners to write successfully in one trial. The refined Java code, which defines function *f*0 is shown below as a member operation of the class F0.

```
public class F0
{
   int res = 0;
   F0() {};

   public int f0(int x)
   {
     switch (x)
      {
        case 0:
          res = 0;
          break;
        default:
        {
          x = x - 1;
          switch (x)
            {
              case 0:
                 res = 0;
                break;
              default:
                { x = x – 1
                   res = f0(x) + 1;
                }
        }}}

      return res;
  }}
```

The above Java class F0 contains the correct operation details code for operation *f*0 after refinement of the previously shown intermediate Java code.

A. Java Class Construction

Each of the HOFL functions defined with **letrec** expression within the constructed HOFL program is translated to a Java function using the rules ⁊. A Java class F0 is defined following the function name *f*0 of the outermost **letrec** function *f*0 defined by expression (2) to encapsulate the Java function *f0* after refinement as a member operation is shown above. The λ-bound variable of the constructed HOFL function *f*0 becomes the formal parameter, i.e., *int x*, of the Java function *f*0, which is defined by translating the HOFL **letrec** function *f*0. In most of the cases, the theorem prover constructs the HOFL program consisting of a single **letrec** expression from each input specification [1]. The class F0 is defined with declaring the constructor F0() and the output variable *res*.

B. Java Application Class Construction

A Java program usually executes by creating objects of the user defined classes within a Java application class containing the *main*() function. To execute the Java operation *f*0 of the class F0 as shown below, we need to build a Java program using a Java application class containing a main function to invoke the function *f*0 by creating object of F0.

```
class F0app
{
static F0 ob = new F0();

public static void main(String args[])
{
   int x = 11;
   System.out.println("Input (x): "+ x);
   System.out.println("Existential Witness (y):"+ ob.f0(x));
 }
}
```

To develop an executable Java program, a Java application class called *F0app* is defined as shown above. The argument *x* of the HOFL **letrec** function call *f*0 *x* used in expression (2) becomes the argument of the Java function call *f*0(*x*) using the object of the class F0 within the *main* function of the Java application class *F0app*. The argument *x* of the HOFL **letrec** function call *f*0 *x* is declared as an integer variable within the *main* function. The operation *f*0 of the class F0 is invoked by passing *x* by initializing an integer number 11 as an argument to the Java function *f*0 by creating an object. The Java function *f*0 computes the same value as the witness value computed by the HOFL program defined by expression (2) for any input value of *x*. The output of executing the constructed Java program with invoking the operation *f*0 with an input 11 is shown below, which computes an existential witness value 5.

```
Input (x): 11
Existential Witness (y): 5
```

The above output computed by the automatically developed Java function *f*0 satisfies the properties defined by the input functional program specification (1).

IX. Implementation and Results

A prototype version of the Java program construction system based on the system architecture shown in Fig. 1 has been tested. The theorem prover Poiti′n [2,3,8] implemented using SML/NJ functional programming language is at the heart of the program construction system. Poiti′n uses a simple higher order functional language (HOFL) with first order quantifiers. The functional program constructed from the input specification is a HOFL **letrec** function, which is output to a disk file for further processing using the PLTS module to generate Java code from this HOFL function. A simple application program has been written using NetBeans IDE Java programming language to implement the rules $\mathcal{T}$ of the PLTS module to translate the constructed HOFL program into semantic equivalent Java code in the form of a Java function, is still under improvement. The generated Java code of the function requires refinement tasks to be performed through using four steps of the refinement phase to build a syntactically correct Java function. The construction of the Java class to encapsulate the generated Java function as a member operation, and the construction of the Java application class for object creation and invoking the member operation are done manually in the current version of the prototype, which is still under improvement. The Java program construction system can be used to generate Java code in all of the cases where the theorem prover Poiti′n is able to construct HOFL program from the constructive proofs of the input program specifications.

A. Validation and Correctness

A number of theorem provers are available besides Poiti′n, e.g., Nuprl, Coq and automatic recursive program synthesis system [4,5,6], which can be used to verify programs, and can be used to construct programs from the proofs of the specifications. Most of the theorem provers and program synthesis systems use axioms or intermediate lemmas and generalizations in order to complete the proof successfully. Poiti′n does not make use of any lemmas, only need generalization to complete the proofs [2]. Hence the number of theorems that can be proved by Poiti′n is also small.

To show that the program construction system shown in Fig. 1 can construct correct Java programs with respect to the input program specification, the following two properties need to be ensured:

- The functional program constructed from the input functional program specification by Poiti′n is correct and satisfies the input specification.
- The mapping rules $\mathcal{T}$ defined for the PLTS module for translating the constructed functional program to equivalent Java code are sound.

The proof of the above two properties is beyond the scope of this paper. The details of the proof of the first property can be found in [2,3,8,9]. To prove the second property, it is sufficient to show that each HOFL construct that is dealt with the rule of $\mathcal{T}$ is translated to equivalent Java code. It is beyond the scope of this paper to give the details of this proof.

B. Examples of Some Program Specifications

Some examples of functional program specifications which can be used to construct functional programs from their constructive proof using Poiti′n are shown below [2].

1) *ANY y:nat.(eqnum x Zero)* $\vee$ *(eqnum x (Succ(y)))*
2) *ANY y:nat.eqnum y (plus x (Succ(Zero)))*
3) *ANY y:nat.(even x)* $\rightarrow$ *(eqnum (double y) x)*
4) *ANY z:nat.(less x y)* $\rightarrow$ *(eqnum(plus x z) y)*
5) *ANYy:nat.or (eqnum(double y) x)*
   (*eqnum*(*Succ*(*double y*)) *x*)

X. Conclusion and Future Work

The approach for Java program construction presented in this paper to solve a particular computational problem uses an inductive theorem prover called Poiti′n [2,3,8]. A HOFL program is automatically constructed from the proof of a functional program specification using Poiti′n, which is translated to a Java function using a PLTS module in order to generate a Java program to get the essence of constructing Java programs from input program specifications. The constructed HOFL program satisfies the input specification. The generated Java function requires refinement to obtain a syntactically correct Java function which can compute the same output as that of the HOFL program [1]. To execute this function, it is encapsulated within a user defined Java class as a member operation, and invoked within a java application class by creating object of the user defined class. The language of Poiti′n is untyped, and hence the input specifications are considered about natural numbers only in the current scope. As far we know from the online literature, for the first time, the approach for the automatic construction of a Java program from the input program specification, i.e. a functional program specification using the constructive proof of the specification is presented in this paper based on the work presented in [1]. The programs are constructed only from the specifications which are proved correct. So, this system constructs correct programs with respect to the specifications. Automatic construction of programs is an interesting area of research in the field of formal software development.

There are a number of directions for continuing further research. First, the Java code generation phase can be improved so that more efficient Java code can be generated, which will require less refinement tasks. Second, the language of the theorem prover Poiti′n can be extended to include type systems [1], and try to handle more difficult specifications for program construction. Finally, the Java class construction phase can be automated to develop an executable Java program.

ACKNOWLEDGMENT

The author thanks all of the people who provided valuable comments to this work.

REFERENCES

[1] Md. Humayun Kabir, Development of a Language Translator for Automatic Construction of Java Programs from Functional Programs, Project Report. Faculty of Mathematical and Physical Sciences, Jahangirnagar University, 2010.

[2] Md. Humayun Kabir, Automatic Inductive Theorem Proving and Program Construction Methods Using Program Transformation. PhD Thesis, School of Computing, Dublin City University, Ireland, September 2007.

[3] G.W. Hamilton and H. Kabir, "Constructing Programs From Metasystem Transition Proofs". Proceedings of the First International Workshop on Metacomputation in Russia, pp. 9-26, 2008.

[4] Seokhyun Han, "Verification of Java Programs in Coq", 2nd Conference on Computer Science and Electronic Engineering (CEEC) 2010, IEEE, pp. 1-8.

[5] F. Loulergue, V. Niculescu, and S. Robillard, "Powerlists in Coq: Programming and Reasoning", First International Symposium on Computing and Networking (CANDAR) 2013, IEEE, pp. 57-65.

[6] A. Armando, A. Smail and I. Green, "Automatic Synthesis of Recursive Programs: the Proof-planning Paradigm", 14th IEEE International Conference on Automated Software Engineering, pp. 2-9, 1997.

[7] Zohar Manna and R. Waldinger, "Synthesis: Dreams $\rightarrow$ Programs", IEEE Transactions on Software Engineering. vol. SE-5, Issue: 4, pp. 294-328, 1979.

[8] H. Kabir and G.W. Hamilton, "Extending Poitin to Handle Explicit Quantification", Proceedings of the 6th International Workshop on First-Order Theorem Proving FTP 2007, pp. 20-34, Liverpool, UK.

[9] G.W. Hamilton, "Distilling Programs for Verification", Proceedings of the 6th International Workshop on Compiler Optimization meets Compiler Verification, ETAPS 2007, pp. 21-35.

[10] Md. Humayun Kabir and G.W. Hamilton, "On Automatic Program Transformation Algorithms", Journal of Electronics and Computer Science, Jahangirnagar University, vol. 10, pp.19-24, June 2009, ISSN 1680-6743.

[11] P. Bertelsen, Compiling SML to Java bytecode, Master's thesis, Department of Information Technology, Technical University of Denmark, January 1998.

[12] Mark Tullsen, Compiling Haskell to Java, Research Report, Department of Computer Science and Engineering, Yale University, New Haven, May 1996.

[13] E. Zieg, "An ATLAS to C Conversion Utility for VDATS", AUTOTESTCON 2008, IEEE, pp. 616-618, Salt Lake City, UT.

[14] H. M Sneed, "Migrating from COBOL to Java", IEEE International Conference on Software Maintenance (ICSM) 2010, pp 1-7, Romania.