PLDI Practical 1

150008859

29/10/2018

1 Usage

Make the following scripts executable.

```
chmod +x clean.sh
chmod +x jasmin.sh
chmod +x run.sh
```

For example to compile and run examples/ExampleA.hs, enter the "prac_1 directory" and run the following command.

```
./clean.sh && runhaskell -i. examples/ExampleA.hs && ./jasmin.sh && ./run.sh
```

The compiler source code can be found in "Main.hs" with Haskell as the choice of language.

The compiled jasmin code is placed in the "output_directory".

2 Completion

Datatype declarations are fully compilable (Bools complete into Integers), however function declarations are partially complete.

Pattern matching can only match constants.

3 Design Decisions

Some minor adjustments have been made to the concrete syntax tree in order to design abstract syntax tree used in my compiler which can found commented in "Main.hs".

3.1 Datatype declarations

Datatype Declarations compile into Class definitions and Super Class definitions in Jasmin. Each class definition features an equals method, toString method and a constructor.

The Jasmin code generated resembles the following Java code, the actual Jasmin code is present in the "examples" folder.

3.1.1 ExampleA.hs

This first example features the datatype Shape which has two constructors accepting primitive datatypes.

```
Datatype Shape = Square Int | Rect Int Int

./clean.sh && runhaskell -i. examples/ExampleA.hs && ./jasmin.sh


Shape.j

Public class Shape
{

}


Square.j

public class Square extends Shape
{
 Integer _1;

 public Bool equals(Object o)
 {

  if (o instanceof Square)
  {
     Square _o_ = (Square) o;

     return _1.equals(_o_._1);
  }

  return false;
 }

}


Rect,j

public class Rect extends Shape
{
 Integer _1;
 Integer _2;
```

```
 public Bool equals(Object o)
 {

  if (o instanceof Rect)
  {
     Rect _o_ = (Rect) o;

     return _1.equals(_o_._1) s&& _2.equals(_o_._2);
  }

  return false;
 }
}
```

3.1.2 ExampleB.hs

This second example demonstrates nesting and recursion, you can create datatype declarations which possess constructors that accept other Datatypes as arguments including itself.

Here is an example of the classical List.

```
datatype List = Cons Int List | Null

./clean.sh && runhaskell -i. examples/ExampleB.hs && ./jasmin.sh


List.j

Public class List
{

}


Cons.j

public class Cons extends List
{
 Integer _1;
 Cons    _2;

 public Bool equals(Object o)
```

```
 {

  if (o instanceof Cons)
  {
     Cons _o_ = (Cons) o;

     return _1.equals(_o_._1) && _2.equals(_o_._2);
  }

  return false;
 }
}
```

```
Null.j

public class Null extends List
{

 public Bool equals(Object o)
 {

  if (o instanceof Cons)
  {
     return true;
  }

  return false;
 }
}
```

3.2 ExampleC.hs

In this final example, I demonstrate mutual recursion which concludes all the features of the datatype declarations.

```
Datatype A = Ctor_A B | A_Int Int
Datatype B = Ctor_B A | B_Int Int

./clean.sh && runhaskell -i. examples/ExampleC.hs && ./jasmin.sh


class A { }
class B { }
```

```
class Ctor_A extends A { B _1; }
class Ctor_B extends A { A _1; }

class A_Int extends A { Integer _1; }
class B_Int extends B { Integer _1; }
```

3.3 Function declarations

Unfortunately, due to my initial incorrect choice of abstract syntax of patterns, I had very little time remaining to complete the practical. In exchange, I shall emit examples of what my compiler can currently do, a copy of my incorrect attempt which be found in the "old_iteration" folder and my strategy including what parts of it I implemented and which parts remain. In addition, I have commented my source code very well which should demonstrate my overall strategy. In addition, the actual Jasmin present in the examples folder shows a more accurate picture.

3.3.1 Patterns

I have no implementation for list of patterns.

To compile constructor patterns, I deviated from the concrete syntax. The concrete syntax for a function declaration is as follows.

```
<decl_b> := id <patt>* = <expr>

<patt>   := id | id <id>* | <constant> | <list_pattern>
```

I adjusted it into the following Haskell data definition

```
data MyPatt = MyPatt_a String | MyPatt_b PCtor | MyPatt_c Const | MyPatt_d [MyPatt]

data Ident  = IdentVar String String | IdentCtr PCtor | IdentCst Const

data PCtor  = PattCtor String [Ident]
```

For example "Rect x 0", is represented as the following.

```
MyPatt_b $ PattCtor "Rect" [IdentVar "x" "Int", IdentCst (INT )]
```

The most notable change here is using a tree like structure to represent the tree like nature of constructor patterns allowing us to pattern match on fields by recursively calling the getfield instruction. In addition, a small, but reasonable adjustment I made to variable identifiers is storing the type, this can be derived from the datatype declarations.

An example of my strategy for compiling a pattern is as following:

```
datatype "Shape" = Rect Int Int | Square Int
datatype "AType" = Ctor Shape Int

f (Ctor (Rect x 0) = x
f (Ctor (Rect x y) = (Rect (x + 1), (y - 1))
```

My pattern matching on constructors involves using instanceof followed by checkcast on the Object arguments, then recursively accessing the fields on the Object or its fields until the target value is reached.

A label is placed between every declaration, if a pattern fails to match you simply jump to that label. On the otherhand, if a pattern is matched, then by the processing of falling through the expression is executed and the function returns. If all the patterns fail to match, then the final expression is executed. Currently, I can only pattern match against Constant values, but not Constructors. Although my recursion is correct, the sequence of bytecode generated is incorrect.

3.3.2 Compiling Functions

Given a set of declarations for a function, the number of patterns represent the arguments to the function. Each function accepts those number of arguments and returns an Object.

For example, a set of declarations with 3 patterns per declaration will compile to the following.

```
"f" Patt_1, Patt_2, Patt_3 = <Ezpr>
"f" Patt_1, Patt_2, Patt_3 = <Ezpr>

public static Object f(Object _0, Object _1, Object _2)
{
```

```
}
```

3.3.3 Compiling Expressions

I can compile all arithmetic expressions, constant expressions, parenthesized expressions. However, I cannot compile the Equality and Greater Than operators, although I emit the correct pattern in my intermediate representation, I did not have time to generate unique labels for the goto statements.

I cannot compile function and constructor expressions, although I believe I have the correct pattern the sequence of instructions generated seems to generate an error.

I have no implementation for list expressions or for case expressions.

Expressions compile into operations on a simple stack machine.

3.3.4 ExampleD.hs

Pattern matching constants and emitting a constant

```
patt
        main : 3
```

```
./clean.sh && runhaskell -i. examples/ExampleD.hs && ./jasmin.sh && ./run.sh
-> 3
```

3.3.5 ExampleE.hs

Pattern matching constants and emitting a constant, while performing arithmetic operations

```
patt
        main : (2 * 3) + (4 / 2)
```

```
./clean.sh && runhaskell -i. examples/ExampleE.hs && ./jasmin.sh && ./run.sh
-> 8
```

3.3.6 ExampleF.hs

```
patt
        datatype Shape = Rect Int Int | Square Int

        main = Rect 3 4
```

```
./clean.sh && runhaskell -i. examples/ExampleF.hs && ./jasmin.sh && ./run.sh
-> (Error) Arguments can't fit into locals in class file Rect
```

Constructor expressions generate errors.