Graphics & Interaction Primitives

Computer Language Processing '15 Final Report

Tommaso Bianchi Fabio Chiusano
EPFL
tommaso.bianchi@epfl.ch fabio.chiusano@epfl.ch

1. Introduction

Builiding a compiler for a language (even the simplest ones) is never a straightforward task and for this reason we splitted up the work in simpler parts.

The first thing we did was to catch the meaningful parts of the source code and store them in objects called tokens. We parsed them later in an abstract sintax tree, which is a faithful representation of the source code (if done correctly). Next we checked such tree in order to find errors and bad coding behavior that the compiler can show to the programmer. While building a compiler, we should take advantage of all possible situations where we can help the programmer and, when possible, show him where something got bad (for instance, showing the line that caused the undesired behavior in the command line) and possibly help him solve the situation. There are a lot of code analysis phases that we could have added, and for the purpose of the course we made two of them. The first one is Name Analysis, whose intent is to check if we have a meaningful sequence of tokens by binding symbols to identifiers and mapping all their occurrences to their declarations, keeping track of the scope of the program. In this way, we can detect a lot of errors, such as variables or classes defined more than once, variables not declared or cycles in the inheritance graph. The second one is Type Checking, that verifies the type safety of a program, making sure that it conforms to all the type rules of the language. The last thing to do was the code generation phase, where we converted our checked abstract syntax tree to bytecode, which can be readily executed by the JVM.

Even though the language we used can be considered simple, it is very powerful (it is Turing complete indeed). However, graphical application developer may not find it well suited for them, as there are no high-level functions that help them. Since we want our programming language to have worldwide success, we decided to ease their life and add elegant primitives for them.

2. Examples

In order to explain our custom extension, we will work through examples[1], from the simplest to the hardest.

Example 1: simple frame

```
object example1 {
  def main() : Unit = { println(new Console().run); }
}

class Console {
  def run(): String = {
    var g: GUI;
    var useless: Int;
    var frameWidth: Int;
    var frameHeight: Int;

    g = new GUI();
    frameWidth = 800;
    frameHeight = 600;
    useless = g.beginGUI("Simple frame",
      frameWidth, frameHeight);

    useless = g.drawString("Hello world!",
      g.getWidth()/2, g.getHeight()/2, 50);

    return "App started";
  }
}
```

[1] The variable *useless* that is present in the examples is necessary due to the fact that we can't return a void type and we can't use expressions as statements

In this example we create a simple frame with a label and we show it to the user. In order to do it, we instantiate a *GUI* object and we call *beginGUI* on it, specifying the frame dimensions and title. It's possible to create more than one frame by simply creating two or more *GUI* objects. The variable that stores the *GUI* object will be used later for all the activities that concern drawing on the frame, manage time and input from keyboard. The method *drawString* is self-explanatory: it takes as arguments the string to show, its coordinates and the font size.

Example 2: moving shapes and time management

```
object example2 {
  def main() : Unit = { println(new Console().run); }
}

class Console {
  def run(): String = {
    var g: GUI;
    var useless: Int;
    var time: Int;
    var frameDuration: Int;
    var rect: Rect;

    g = new GUI();
    useless = g.beginGUI("Simple frame", 800, 600);

    rect = new Rect().init(new Point().init(10, 10), 20, 100);

    time = 0;
    frameDuration = 1;

    while(true){
      useless = rect.draw(g);
      if(frameDuration < g.getTime() - time){
        useless = g.clear();
        useless = rect.move(1, 1).draw(g);
        time = time + frameDuration;
      }
    }

    return "App started";
  }
}
```

Here we create a frame with a moving rectangle on it, instantiating a *Rect* object. Calling *init* on a *Rect* object, we give it its top left point, a width and a height. A point is represented by the *Point* class whose fields are obviously its coordinates (the coordinate system has the origin in the top left corner). At last, we move the rectangle. This is accomplished by painting it at each execution of the loop and moving it each time the frame duration has passed. The *clear* method clears the frame from everything that was previously painted and the *draw* methods draw the object that they are called on. At last, the *getTime* method returns the time elapsed from the creation of the *GUI* object and the *move* method increments the coordinates of each point of the rectangle by its arguments.

Example 3: Math and Frac for complex movement

```
object example3 {
  def main() : Unit = {
    println(new Console().run());
  }
}

class Console {
  def run(): String = {
    var g: GUI;
    var useless: Int;
    var time: Int;
    var frameDuration: Int;
    var frameWidth: Int;
    var frameHeight: Int;
    var square: Rect;
    var squareSide: Int;
    var c: Point; // movement center
    var r: Int; // movement radius
    var math: Math;

    g = new GUI();
    frameWidth = 800;
    frameHeight = 600;
    useless = g.beginGUI("Simple frame",
      frameWidth, frameHeight);

    squareSide = 20;
    c = new Point().init(frameWidth/2, frameHeight/2);
    r = frameHeight/2 - 50;
    square = new Rect().init(new Point().init(c.x() -
      squareSide/2 + r, c.y() - squareSide/2), squareSide,
      squareSide);

    time = 0;
    frameDuration = 1;
    math = new Math();

    while(true){
      useless = square.draw(g);
      if(frameDuration < g.getTime() - time){
```

```
      useless = g.clear();
      useless = square.setCenter(new Point().init(
        c.x() + math.cos(g.getTime()).times(
          new Frac().init(r, 1)).getRoundedInt() -
          squareSide/2,
        c.y() + math.sin(g.getTime()).times(
          new Frac().init(r, 1)).getRoundedInt() -
          squareSide/2
      )).draw(g);
      time = time + frameDuration;
    }
  }

  return "App started";
  }
}
```

Here we show how to use the classes *Math* and *Frac*, that help us with more complex computations. This is a little more challenging since in Tool there is no float type and therefore we have to rely only on integers. A square is created using the *Rect* class and is moved in circular motion around the center of the frame thanks to the *sin* and *cos* functions of the *Math* class, that return a *Frac* (i.e. our representation of floats) and are in turn rounded to the nearest integer using the *getRoundedInt* function. The *setCenter* method of the *Rect* class centers the square to the point that is passed as argument.

Example 4: keyboard control

```
object example4 {
  def main() : Unit = {
    println(new Console().run());
  }
}

class Console {
  def run(): String = {
    var g: GUI;
    var useless: Int;
    var time: Int;
    var frameDuration: Int;
    var square: MySquare;

    g = new GUI();
    useless = g.beginGUI("Prova 3", 800, 600);

    square = new MySquare().init(
      new Point().init(10, 30), 20);

    time = 0;
    frameDuration = 1;
    while(true){
      useless = square.draw(g);
      if(frameDuration < g.getTime() - time){
        useless = g.clear();
        useless = square.move(g).draw(g);
        time = time + frameDuration;
      }
    }

    return "App started";
  }
}

class MySquare {
  var square: Rect;
  var sideLength: Int;

  def init(topLeft: Point, sL: Int): MySquare = {
    square = new Rect().init(topLeft, sL, sL);
    sideLength = sL;
    return this;
  }

  def move(g: GUI): MySquare = {
    var dirX: Int;
    var dirY: Int;

    if(g.getKey(119)) // w, go up
      dirY = 0-1;
    else if(g.getKey(115)) // s, go down
      dirY = 1;
    else
      dirY = 0;
    if(g.getKey(97)) // a, go left
      dirX = 0-1;
    else if(g.getKey(100)) // d, go right
      dirX = 1;
    else
      dirX = 0;

    square = square.move(dirX, dirY);

    return this;
  }

  def draw(g: GUI): Int = {
    var useless: Int;
    useless = square.draw(g);
    return 0;
  }
}
```

In this example we create a small square and we move it on the screen with the use of the keyboard. The methods used are almost the same of example 2, except for the *move* method of the *MySquare* class. In its implementation, we check if the keys "w", "a", "s" and "d" are pressed with the method *getKey* of the GUI class, that returns *true* if such case (it wants the Ascii code of the character as argument). Later we move the square calling the method *move* on it.

More on new primitives
We have added six classes that are always in scope:

- **GUI**
  It creates a frame with the method *beginGUI*. It also manages the time elapsed with *getTime*, the keyboard with *getKey* and the mouse with *getMousePosition* and *getMouseButton*.
- **Point**
  Basic *Point* class with coordinates and methods, such as:
  - *plus*, that adds it to another point (as they were vectors);
  - *times*, which is for scalar multiplication;
  - *draw*, self-explanatory;
  - *drawLine*, that draws a line that links two points;
  - *move*, that moves it of some dx and dy;
  - *isInside*, that returns true if such point is inside a *Rect*.
- **Rect**
  A rectangle defined by its top left point, width and height. It has methods to obtain all its vertices, to draw it and to move it.
- **Circle**
  A circle defined by its center and its radius.
- **Frac**
  A class that represents a fraction, with a lot of operations that can be done on it.
- **Math**
  A class that contains a lot of useful operations on integers and fractions, mainly used for graphic computations.

Extra: Pong
We've built a game like Pong to give an example of a more complex program that uses the new primitives. It's in the *programs* directory along with all the other examples, we invite who wants to try it! Left player uses "w" and "s", while right player uses "o" and "l".

3. Implementation

3.1 Theoretical Background

We didn't use any theoretical concepts for the implementation of graphic and user interaction primitives.

3.2 Implementation Details

As showed in the example section, the programmer sees some new always in-scope classes that help him with everything related to graphic and user interaction. We're going to explain their implementation in each phase of our compiler.

- **Lexer**
  In this phase we analyze not only the source code, but also the extra classes that the programmer can use. These extra classes are stored in the compiler and is its job to take care of them. However, the methods of the GUI class are not implemented in Tool, but they will be implemented later in the code generation phase and therefore the compiler won't look for a Tool body. This is done through the use of a new keyword (and so a new token): *native*. The GUI class therefore looks like this:

```
class GUI {
  native def beginGUI(name: String, width: Int,
    height: Int): Int
  native def clear(): Int
  native def getWidth(): Int
  native def getHeight(): Int
  native def getTime(): Int
  native def getKey(id: Int): Bool
  native def drawLine(x1: Int, y1: Int, x2: Int,
    y2: Int): Int
  native def drawString(str: String, x: Int,
    y: Int, fontSize: Int): Int
}
```

Their implementation is done, as shown later, through the use of some Java graphics libraries. We have decided to implement only the most basic ones in this way: all the other methods of the always in-scope classes related to graphics and user interaction rely on such methods of the GUI class. For example, the *drawLine* method of the *Point* class draws a line between two points using the native *drawLine* method of the *GUI* class:

```
def drawLine(g: GUI, other: Point): Int = {
  var useless: Int;
  useless = g.drawLine(myX, myY,
    other.x(), other.y());
  return 0;
}
```

To put it in a nutshell, at the end of the source code there will be some other code lines that look like this:

```
class GUI {
  // Native methods
}
class Point {
  // Some methods built on the native one of the
  // GUI class, such as draw() and drawLine(otherPoint)

  // Some helper methods, such as plus(otherPoint)
  // and isInside(rect)
}
class Rect {
  // Some methods
}
// Other classes
```

- **Parser**
  Since there is a new type of method (i.e. a *native* method with no Tool implementation), we need a new abstract tree to represent it:

```
case class NativeMethodDecl(
  retType: TypeTree,
  id: Identifier,
  args: List[Formal])
  extends Tree with Symbolic[MethodSymbol]
```

Basically it has all the fields that a *MethodDecl* has, except for the ones related to its body. The *ClassDecl* class now looks like the following:

```
case class ClassDecl(
  id: Identifier,
  parent: Option[Identifier],
  vars: List[VarDecl],
  methods: List[MethodDecl],
  nativeMethods: Option[List[NativeMethodDecl]])
  extends Tree with Symbolic[ClassSymbol]
```

- **Code Analysis**
  In this phase there is a little new. In Name Analysis, we have to attach types to the native methods, just like we did with normal method declarations. In Type Checking, everything is the same as it was, that is we check the type safety of the program.

- **Code Generation**
  This is the hardest phase and therefore where most of the changes happened. We remind now how the *GUI* class was structured:

```
class GUI {
  ...
  native def getHeight(): Int
  native def getTime(): Int
  ...
}
```

Now, instead of manually implement the native methods in bytecode, we[2] found that the simplest way to do it was to just let the *GUI* class be a subclass of a Java class (i.e. ToolGUI.java) that has those methods implemented in Java. This is how we create the class file of the *GUI* class:

```
if (classDecl.id.value == "GUI")
  new ClassFile(classDecl.id.value, Some("ToolGUI"))
```

The ToolGUI.java file can be found in the gui_lib directory of the compiler and it contains the ToolGUI class, whose functions are implemented with the help of the Java Awt and Swing graphic libraries. In this way, when we implement the native methods, we can simply call their equivalent from the parent class:

```
InvokeSpecial("ToolGUI", methodDecl.id.value,
  methSignature)
```

The last thing to do in order to make all the native methods work is to have the class files of ToolGUI.java in the destination folder. We accomplish this by compiling the ToolGUI.java file (with the use of scala.sys.Process and javac) and putting it into the destination directory. This code is located in Main.scala.

A brief look at ToolGUI.java

This file is where the native methods are implemented. The class has four private fields:

1. a frame, that is used in most of its methods;
2. an array of 256 booleans, whose *i-th* cell is true if the key whose character has Ascii code *i* is pressed in that moment;

[2] With some help from Manos Koukoutos

3. an array of two booleans, where the first cell is for the mouse left button and the second one is for the right button. They behave like the cells of the keyboard array;
4. the time elapsed from the calling of *beginGUI*.

All the methods are straightforward except for *beginGUI*. Here we create a frame and we set its title and dimensions. Then, we add three listeners to the frame:

1. a *WindowAdapter*, that allows to terminate the process as soon as the frame is closed;
2. a *MouseAdapter*, that for now only prints which button was pressed;
3. a *KeyAdapter*, that manages the keys array.

At last, we start a *Timer* that updates the *time* field each 10 milliseconds and we make the frame visible.

Some small changes in Positioned

Even though the programmer is not supposed to modify the *GUI* library, it can be useful to us, the compiler developers, to easily find errors in such library by showing them in the command line. This is possible thanks to a modification of the *setPos* method of the *Positioned* trait, whose body is now the following:

```
val lib = new File("./gui_lib/GUI.tool")
val t1 = Source.fromFile(file).mkString
val linesFile1 = t1.lines.size

if (lineOf(pos) <= linesFile1) { /* If the
  position is in the file in input */
  _line = lineOf(pos)
  _col = columnOf(pos)
  _file = Some(file)
}
else { // If the position is in the library GUI
  _line = lineOf(pos) - linesFile1
  _col = columnOf(pos)
  _file = Some(lib)
}
```

In those lines we check if the position is in the input file or in the *GUI* library by comparing its line to the number of lines of the input file and we set the *_file* variable accordingly.

4. Possible Extensions

There are a lot of possible ways to extend our compiler for graphic and user interaction. For instance, we can:

- add new graphic native methods by simply adding them to ToolGUI.java and GUI.tool, such as some for choosing colors and filling shapes. That would imply then to recompile ToolGUI.java;
- extend the Tool graphic library with more high-level classes and functions with more complex geometric shapes and actions, such as an action that makes a shape move in circular motion, as shown in example 3. In order to implement complex shapes, using an array of *Point* would be the easiest way but two arrays of integers (i.e. one for abscissas and one for ordinates) can be used too;
- add a float type, that would make easier to implement complex computations without relying on the *Frac* class;
- add primitives to display bitmaps.
- extend the language to allow functions to return *Unit*. That would make the code much more cleaner.