Amy language: C Backend

Final Report

Jeniffer Lima Graf   Alexander Sanchez de la Cerda

EPFL

{firtname.lastname}@epfl.ch

1. Introduction

Describe in a few words what you did in the first part of the compiler project (the non-optional labs), and briefly say what problem you want to solve with your extension.

This section should convince us that you have a clear picture of the general architecture of your compiler and that you understand how your extension fits in it.

2. Examples

Give code examples where your extension is useful, and describe how they work with it. Make sure you include examples where the most intricate features of your extension are used, so that we have an immediate understanding of what the challenges are.

```
Amy:
object Hello {
  Std.printString("Hello " ++ "world!")
}

becomes in C:
#include <stdio.h>
#include "std.h"

int Hello_main() {
  Std_printString(String_concat("Hello ", "world!"));
  return 0;
}
```

```
One line of Amy code can generate a lot of C Code

case class Fractional(num: Int, denom: Int) extends Rational

becomes in C:
typedef struct Fractional {
  int param0; // numerator
  int param1; // denominator
} Fractional;

Rational Fractional_Constructor(int param0, int param1)
{
  Fractional* fractional = malloc(sizeof(Fractional));
  fractional->param0 = param0;
  fractional->param1 = param1;

  Rational rational = malloc(sizeof(Abstract_Rational));
  rational->instance = fractional;
  rational->caseClass = 1;

  return rational;
}

#define instance_fractional(abstr_class)((Fractional*)abstr_class->
```

This section should convince us that you understand how your extension can be useful and that you thought about the corner cases.

3. Implementation

This is a very important section, you explain to us how you made it work.

3.1 Theoretical Background

If you are using theoretical concepts, explain them first in this subsection. Even if they come from the course (eg. lattices), try to explain the essential points *in your own words*. Cite any reference work you used like this [Appel 2002]. This should convince us that you know the theory behind what you coded.

3.2 Implementation Details

The hardest part was finding a working and readable substitution for Amy's abstract and case class (which extend the abstract class) functionality. How can we have "generics" and "inheritance" in C: that seemed like an impossible challenge. So we contemplated on how to achieve this and sat down and tried to come up with ways

4. Possible Extensions

If you did not finish what you had planned, explain here what's missing.

There would be several possible extensions and we might continue working on it because simple Amy-Code is easily translated to good usable C code. The biggest possible extension is naming of our structures and types. Our CaseClass parameters are currently only named in a running sequence of indices. This is due to the fact that in the pattern matching the user defines the "variable names" and they may not necessarily match the case parameters. This leads to the main extension which is having a big map that keeps track of the modules and which abstract class holds which case classes and in turn to which parameters they have.

It is not a huge thing to do but with all of our other exams this is something we leave for when we revisit the code next time.

Conveniently would also be to move the function signatures that are generated into a separate header(.h) file which is then imported with #include "library.h". Along these lines we could also instead of concatenating everything into a file, generate a folder, convert all the files from sbt run <modules.scala* main.scala> individually to C and then import the modules into the main file (last one) and run that file (main).

Another cool idea which only loosely fits with the description but would be very useful in achieving better readability would be augmenting the lexer with saving comments right before a def and save them in a map with the name after the def. We could then map back part of the original documentation and perhaps that would serve having better readable C code (when the programmer followed good comment practice describing what the function does rather than how).

One thing we not fulfilled upon was our promise to deliver C code that does not produce warnings [but in a positive way]. We discovered during our journey that when a user does not suppress a parameter in a pattern match that is not needed that C will issue a warning of an unused variable. And when you think about it this is correct it is not a warning generated because of a bad code translation but a genuine warning that makes sense. So our C backend project actually helped me improve my example code written in the very first lab! I could replace the not used parameters with wildcards and Amy code still works. So our C backend project was not only code translation but also created a warning system for Amy code.

Access restriction, allocation and const

switch

promise no warnings

h files with function signatures

5. Appendix

5.1 Standard Library

The std is included with every document (C file) and brings over some standard Amy functionality.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <stdbool.h>

char* String_concat(const char* s1, const char* s2);
void Std_printString(char* string);
void Std_printInt(int integer);
char* Std_digitToString(int digit);
void Std_printBoolean(bool b);
char* Std_intToString(int i);
char* Std_booleanToString(bool b);

char* String_concat(const char* s1, const char* s2)
{
  const size_t len1 = strlen(s1);
  const size_t len2 = strlen(s2);
  char* result = malloc(len1 + len2 + 1);
  memcpy(result, s1, len1);
  memcpy(result + len1, s2, len2 + 1);
  return result;
}

void Std_printString(char* string)
{
  printf("%s\n", string);
}

void Std_printInt(int integer)
{
  printf("%d\n", integer);
}

char* Std_digitToString(int digit)
{
  char* string = malloc(sizeof(int));
  sprintf(string, "%d", digit);
  return string;
}

void Std_printBoolean(bool b)
{
  Std_printString(Std_booleanToString(b));
}

char* Std_intToString(int i)
{
  if (i < 0) {
    return String_concat("-", Std_intToString(-(i)));
  } else {
    int rem = i % 10;
    int div = i / 10;
    if (div == 0) {
      return Std_digitToString(rem);
    } else {
      return String_concat(Std_intToString(div),
        Std_digitToString(rem));
    }
  }
}

char* Std_booleanToString(bool b)
{
  if (b) {
    return "true";
  } else {
    return "false";
  }
}
```

References

A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 2nd edition, 2002.