Amy language: C Backend

Final Report

Jeniffer Lima Graf   Alexander Sanchez de la Cerda

EPFL

{firtname.lastname}@epfl.ch

1. Introduction

In the first part of the project we created an interpreter, a lexer, a parser, a name analyzer and a type checker for the amy language. After the stage of the type checker we have basically checked that we only call declared and accessible variables and functions, that we have no duplicates, that we only assign valid types to each other and that there are no forbidden operations like division by 0 in our code. Also the order of the natural left association and lazy evaluation should be taken care of, so the only thing missing was a generator. At the last stage of the until now implemented compiler we have defined a code generator which translates Amy code and produces WebAssembly output.

One can imagine that the generated WebAssembly files are not perfectly readable and even less understandable for the human eye. It was also quite a challenge to reason about the postfix language and actually implement the code generation, especially the pattern matching. The C language, which is considered as the next more advanced language on top of the Assembly language would be much more human friendly. We can read C files and easily reason about them. Functions have names and there are types something we do not have in assembly. So thanks to our C backend we can now read sleek (3 line Hello World! program), simple amy code that has several elements of functional languages and then just cross it over to C and continue there; implement things that are perhaps not possible or collaborate with somebody who has never seen amy language before. And if we want assembly we can also have that: one just invokes ones favorite C compiler with the assembly flag and one receives the target machine instructions.

2. Examples

It is a bit hard to give examples here because our project was not so much about improving a specific aspect of the Amy language but rather adding a new dimension to it. We produce human readable, well-known – C code.

The C source files are simply generated by running the desired files similarly how we did it for WebAssembly. this time instead of /wasmout one receives a /cout folder with the translated source file inside. Some nice examples to try.

Perhaps my cool Fractional implementation in Amy (take a look) which is translated to C code (take a look, C but with so much more functionality). Who never wanted rational arithmetic in C?:

run library/Option.scala library/List.scala library/Std.scala examples/Fractional.scala

Or perhaps just try out the Factorial implementation:

run library/Std.scala examples/Factorial.scala

Playing Hanoi:

run library/Std.scala examples/Hanoi.scala

Or just keeping it simple with Hello World?

run library/Std.scala examples/Hello.scala

Generically one has following syntax for producing a C file called main.c with all the aforementioned modules included:

run <modules.scala* main.scala>

```
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->instance)
```

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.

For the pattern matching we went on with the matchAndBinding function approach which returns a boolean and the new defined locals for and individual case, as we have seen in the course. With the boolean we evaluate, if the specific case is the one we should truly be matching on. The condition also includes the parameters. So basically for a wildcard and an id pattern it is always true. For the id pattern, the id is then additionally added to the list of locals, associating the expression that is being matched on with the this newly defined id. For the Case Classes we check that it's the same Class for the given type and the the literals we match directly on the matching expression.

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

In the Utils file we created a trait "CType" and an implicit def which converty types from the TreeModule to CTypes. This is needed because for C every variable declaration, every function parameter and every Function has a type or return type.

We made sure that in the beginning of the module printer all the function declarations are printed, such that we can call on functions before we implement them. The stdio, stdlib, string and stdbool header files are also included in every module such that builtin functions like scanf and printf and also the bool types can be easily used in the code generation.

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.

Another possible extension could be improved translation of code. Better access restriction in our C code by making our variables, constants. Also we could better allocate and free memory. Which would be tricky because we have to keep track of all of our allocation and determine when to free them (non-trivial).

One addition we had partially done was having an enum that maps all case classes to enums and the goal would have then been if we encounter a CaseClassPattern to actually use a switch instead of a simple if. This is not yet finalized but would be a great and not too hard starting point for picking off again the project.

```
typedef enum {NIL, CONS} L_LIST;

which would have allowed:

int head(L_List l) {
        switch(l->caseClass) {
                case CONS:
                        return inst_cons(l)->head;
                case NIL:
                        fprintf(stderr, "%s", "head(Nil)");
                        [...]
        }
}
```

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.