C Backend Extension

Computer Language Processing '21 Final Report

Alexis Schlomer Adrien Nelson Rey
EPFL
alexis.schlomer@epfl.ch adrien.rey@epfl.ch

1. Introduction

1.1 Our Original Compiler

Our original Amy compiler generates an executable WebAssembly file from user written Amy code. This is done in a 5-stage pipeline composed of respectively:

1. A *Lexer* that transforms the stream of characters of our Amy source code into a stream of meaningful tokens. Regular expressions are used to find these appropriate tokens using the longest match rule.
2. A *Parser* which transforms this sequence of tokens into an Abstract Syntax Tree (AST), containing the logic of our program. This is done using a LL(1) left-associative grammar.
3. A *Name Analyser* responsible for binding each identifier to its appropriate declaration. Modules, functions, constructors and variables are now mapped into a significant symbol table.
4. A *Type Checker* to help the programmer detect type inconsistencies by inferring the expected type at every level and comparing it with the actual type using the Hindley-Milner algorithm.
5. A *Wasm Code Generator* that finally creates the right executable WebAssembly files. Expressions are transformed into the Reverse Polish Notation (RPN) to satisfy Wasm stack based machine.

1.2 Our C Backend Extension

The goal of the C backend extension is to provide an option to generate, from the same Amy source code, compilable C files that have the same behaviour than our original Amy program when executed.
Concretely this means adapting the fifth stage of our pipeline by letting it generate executable C source files instead of WebAssembly code.

2. Examples

2.1 Subset Sum

Our first example is a little program of the well known subset sum problem on an integer interval.
It asks the user to enter the desired sum and the integer bounds of an interval $[a, b]$ and returns the number of subsets in this interval that adds up to the given sum.
For example if we input 10 and $[2, 5]$ the program will output 7 as we can decompose our interval into the following sets:

$$\{(5,5),(5,3,2),(4,4,2),(4,3,3),$$
$$(4,2,2,2),(3,3,2,2),(2,2,2,2,2)\}$$

Here is the Amy program:

```
object SubsetSum
    fn subsetSum(a: Int(32), b: Int(32),
                 sum: Int(32)): Int(32) = {
        if(b < a || sum < 0) {
            0
        } else {
            if(sum == 0) {
                1
            } else {
                subsetSum(a, b, sum − b)
                    + subsetSum(a, b − 1, sum)
            }
        }
    }
    Std.printString("Enter the Sum :");
    val sum: Int(32) = Std.readInt();
    Std.printString("Enter lower bound of interval :");
    val a: Int(32) = Std.readInt();
    Std.printString("Enter upper bound of interval :");
    val b: Int(32) = Std.readInt();
    Std.printInt(subsetSum(a, b, sum))

end SubsetSum
```

This program illustrates that our extension works for the following central Amy functionalities:

1. Recursive functions
2. If-Clauses
3. Strings generation and utilisation
4. I/O library

Here is the generated C code for this program:

```c
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include "std.h"
#include "error.h"
/*
 ...
 Imported Std library functions (see in cout directory)
 ...
*/
int32_t SubsetSum_subsetSum(int32_t a, int32_t b, int32_t sum) {
 int32_t ite_4;
 if(((b < a) || (sum < 0))) {
  ite_4 = 0;
 }
 else {
  int32_t ite_3;
  if((sum == 0)) {
   ite_3 = 1;
  }
  else {
   ite_3 = (SubsetSum_subsetSum(a, b, (sum - b)) +
     SubsetSum_subsetSum(a, (b - 1), sum));
  }
  ite_4 = ite_3;
 }
 return ite_4;
}

int main() {
 char* string_5 = malloc(strlen("Enter the Sum :") + 1);
 strcpy(string_5, "Enter the Sum :");
 Std_printString(string_5);
 int32_t sum_0 = Std_readInt();
 char* string_6 = malloc(strlen("Enter lower bound of interval :")
     + 1);
 strcpy(string_6, "Enter lower bound of interval :");
 Std_printString(string_6);
 int32_t a_0 = Std_readInt();
 char* string_7 = malloc(strlen("Enter upper bound of interval :")
     + 1);
 strcpy(string_7, "Enter upper bound of interval :");
 Std_printString(string_7);
 int32_t b_0 = Std_readInt();
 return Std_printInt(SubsetSum_subsetSum(a_0, b_0, sum_0));
}
```

This code is compilable in C if passed with both libraries *error.h* and *std.h*. The *subsetSum* function implementation is nearly straightforward (except for the usage of the variables *ite_i* which will be explained later in the implementation details). The *main* function is a bit more cumbersome as a result of allocating all the strings.
Keep in mind when using this program that subset sum is a NP-complete problem and will take an eternity for big numbers.

2.2 List Functions

Our second example is a program that uses some functions on lists. These functions are *reverse* which reverses a list, *mergeSort* which sorts it (both from the List library) and a new function *MovingAverage* which computes the moving average.

Here is the Amy program:

```
object MovingAverage

fn movingAverage(list: L.List, sum: Int(32),
                 count: Int(32)): L.List = {
    list match {
      case L.Nil() => L.Nil()
      case L.Cons(h, t) =>
        val average: Int(32) = (h + sum) / (count + 1);
        L.Cons(average,
          movingAverage(t, sum + h, count + 1))
    }
}

val list: L.List = L.Cons(5, L.Cons(2, L.Cons(7,L.Cons(4,
                   L.Cons(1, L.Cons(6, L.Nil()))))));
Std.printString(L.toString(L.mergeSort(list)));
Std.printString(L.toString(L.reverse(list)));
Std.printString(L.toString(movingAverage(list, 0, 0)))

end MovingAverage
```

This program illustrates that our extension works for the following central Amy functionalities:

1. Pattern matching
2. Algebraic data structures
3. List and Option library

Here is the generated C code for this program:

```c
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include "std.h"
```

```c
#include "error.h"

/*
 ...
 Other library functions (see in cout directory)
 ...
*/
void* MovingAverage_movingAverage(void* list, int32_t sum,
      int32_t count) {
 void* match_23;
 if((*(int32_t*)list) == 0) {
  struct L_Nil* struct_20 = malloc(sizeof(struct L_Nil));
  struct_20->constr_index = 0;
  match_23 = struct_20;
 }
 else {
  if((*(int32_t*)list) == 1 && 1 && 1) {
   int32_t h_11 = ((struct L_Cons*)list)->att0;
   void* t_9 = ((struct L_Cons*)list)->att1;
   int32_t average_0 = ((h_11 + sum) / (count + 1));
   struct L_Cons* struct_21 = malloc(sizeof(struct L_Cons));
   struct_21->constr_index = 1;
   struct_21->att0 = average_0;
   struct_21->att1 = MovingAverage_movingAverage(t_9, (sum
     + h_11), (count + 1));
   match_23 = struct_21;
  }
  else {
   Error_error("Match error!");
  }
 }
 return match_23;
}

int main() {
 struct L_Nil* struct_28 = malloc(sizeof(struct L_Nil));
 struct_28->constr_index = 0;
 struct L_Cons* struct_27 = malloc(sizeof(struct L_Cons));
 struct_27->constr_index = 1;
 struct_27->att0 = 6;
 struct_27->att1 = struct_28;
 struct L_Cons* struct_26 = malloc(sizeof(struct L_Cons));
 struct_26->constr_index = 1;
 struct_26->att0 = 1;
 struct_26->att1 = struct_27;
 struct L_Cons* struct_25 = malloc(sizeof(struct L_Cons));
 struct_25->constr_index = 1;
 struct_25->att0 = 4;
 struct_25->att1 = struct_26;
 struct L_Cons* struct_24 = malloc(sizeof(struct L_Cons));
 struct_24->constr_index = 1;
 struct_24->att0 = 7;
 struct_24->att1 = struct_25;
 struct L_Cons* struct_23 = malloc(sizeof(struct L_Cons));
 struct_23->constr_index = 1;
 struct_23->att0 = 2;
 struct_23->att1 = struct_24;
 struct L_Cons* struct_22 = malloc(sizeof(struct L_Cons));
 struct_22->constr_index = 1;
 struct_22->att0 = 5;
 struct_22->att1 = struct_23;
 void* list_0 = struct_22;
 Std_printString(L_toString(L_mergeSort(list_0)));
 Std_printString(L_toString(L_reverse(list_0)));
 return Std_printString(L_toString(
      MovingAverage_movingAverage(list_0, 0, 0)));
}
```

Note that this generated C code is much longer (692 lines) since it includes all functions from the Std, List and Option library. Hence, we decided to only show the interesting parts.
We can see here how ADS are generated in the *main* function and how they are passed to another function (using a void pointer trick which will be explained later in the implementation details). We can also observe how pattern matching is carried out using a number of if-clauses in the *movingAverage* function.
Note: We didn't show *mergesort* and *reverse* here but we invite you if interested to look it up in *MovingAverage.c* line 363 and line 133 respectively.

3. Implementation

Our extension is quite rich and has a lot of features, as we made it work for any arbitrary Amy code. Hence covering all topics would make the report unreasonably long. Thus, we decided to only develop the most crucial aspects of our extension.
We will first mention some important theoretical background concerning the C language before diving into the implementation details of the most important features.

3.1 Theoretical Background

3.1.1 Statements VS Expressions

As opposed to Amy, C is a highly imperative language. This means that it does not naturally support nested expressions such as:

```c
if(1; /*More nested expressions;*/ 1 == 1) {
  // Any expression
} else {
  // Any expression
}
```

The reason being that the main granularity of C is composed of statements like in Assembly. These statements do not return any value, they solely generate side effects that can be used by subsequent statements. This is in strong opposition to Amy, where the main granularity are expressions that by nature return a value.

In the C language, the above code snippet indeed doesn't even return a value. It cannot be evaluated like in Amy.
In order to overcome this problem, we need to understand what aspects in C are functional (ie: behave like in Amy). We will qualify this code as *pure expressions*, as they only return a value without any side effects.

$$IntLiterals/Pointers \quad UniOp(!, -)$$

$$Variables \quad TernaryOp(\_ ? \_ : \_)$$

$$BinOp(+, -, *, /, \%, ==)$$

Other code such as declarations, initialisations, allocations and string concatenations do not return any value as opposed to these *pure expressions*. Yet, they are still needed as without them it would be for example impossible to declare and (re)assign any variable.
Thus, our goal is to express our whole program using a composition of *pure expressions* that will get returned, preceded by prerequisite code that handles the correct declarations, initialisations and other side effects needed by the *pure expression*.
This specific mechanism constitutes the hearth of our code generation mechanism, and is incarnated by the *cgReturnExpr* function, which we will develop below.

3.2 Architectural Changes

In order to implement our extension, we have to perform some important architectural changes. We mainly reused the Wasm Code Generator as it shares a lot in common with our extension. Both take as input the same program associated with its symbol table and output a stream of character. Yet we still have to change a few notable parts.

3.2.1 Instructions VS Tokens

In Wasm, we could use as indivisible unit of code a succinct instruction. Thus, code was simply defined as a sequence of such instructions. This is no longer possible for a C backend, as we are dealing with a higher-level programming language. It's indivisible unit becomes a token, which is consistent with what we did in the first stage of the pipeline. We can once again characterise code as a stream of tokens, with the particularity that we now have to generate code from tokens, and not extract tokens from code as we did in the lexer.

3.2.2 File Printer

The printing phase must also be adapted, as the nature of the generated file is radically different. Additionally to the *mkFun* and *mkCode* functions, we need to add three functions to get a complete C file:

- *mkImport* Imports the correct libraries at the start of every file.
- *mkStruct* Declares all required structures at the start of the program so that they can be used in the code.
- *mkSign* Generates the correct function prototypes to avoid cyclic function dependencies and order sensitive function initialisations.

3.2.3 Other Useful Changes

The nature of the Function class needs to be adapted to correspond better to what is required from a C point of view. This means tracking the exact function signature and typing additionally to its code. Furthermore, we need to add a Struct class to represent C structures as they are generated analogously to functions. Finally, we adapted the default imports and coded our own C libraries for I/O functions and the error crashing function since it does not inherently exist in C (see *std.h* and *error.h*).

3.3 Implementation Details

Here we will describe the five main difficulties we encountered while coding the *CodeGen* file and how we achieved to fix them.

3.3.1 cgExpr and cgReturnExpr

cgExpr signature is quite different from the WebAssembly generation. As mentioned before, we now need to generate not one unique code but a pair of code. One code for the pure expression and another one for the prerequisites. Thus, *cgExpr* returns a *PureExpr* type which is a triplet of (*Code, Type, Code*), representing respectively (Pure expression code, Pure expression type, Prerequisite code for the pure expression).
This *cgExpr* function is then called by the *cgReturnExpr* function, responsible for creating a unique code in the form :

```
1 prerequisites; // All the prerequisites of the entire function
2 return pure_expression;
```

3.3.2 Strings

Strings can cause some problems in C because of their scope. Indeed, strings scope in C is limited to their block of code. If we return a static string created in the body of a function it will not be valid anymore. We thus made the choice to allocate dynamically all strings using the *malloc* function. This way all strings can be used everywhere in the program without worrying about their scope. Then for each string in the program its initialisation is in its prerequisite and the pure expression is a variable. Here is an example for the string "Hello world!"

```
// Prerequisite
  char* string_0 = malloc(strlen("Hello world!:") + 1);
  strcpy(string_0, "Hello world!");
// Pure Expression
  string_0;
```

3.3.3 Abstract Data Structures

We made the logical choice to implement Abstract Data Structures using C structures. For that, each ADS will be represented by a structure with its constructor index as an int_32 as its first field (for pattern matching) followed by its parameters. For example the simple case class $Human(name : String, age : Int(32))$ is implemented as follows:

```
struct Human {
int32_t constr_index;   // Constructor index
char* att0;             // Name
int32_t att1};          // Age
// Notice how we forgot the old names.
// More on this in the possible extensions section.
```

A Problem encountered when working on ADS was how to deal with genericity in C. Assume the following Amy code:

```
abstract class Fruit

class Apple(price:Int(32)) extends Fruit
class Banana(price:Int(32)) extends Fruit

fn f(fruit:Fruit):Int(32) = ...
```

How to handle f parameter since Fruit is not a struct in C and needs to accept both *struct Apple** and *struct Banana** at the same time. A simple solution is to use a void pointer, which will accept both previous types simultaneously. Note that we can be sure that it will only accept these two types only by relying on Amy's type checker.

Then the f signature becomes:

```
int32_t f(void* fruit_0) {...}
```

The use of a void pointer can be generalised to the other uses of ADS in the program. As such, ADS (except for their initialisation) are always represented by a void pointer.

3.3.4 If-statement

As a first attempt to implement if-statements we used the ternary operator $(\_?\_ : \_)$. This was the choice we proposed during our presentation in December. At first it seems like a good idea since it stays a pure expression and can thus be directly used. However this operator cannot evaluate the prerequisites lazily, which is prompt to bugs. Here is an example:

```
fn rec(n: Int(32): Int(32) = {
    if(n==0){1} else {
            val a: Int(32) = rec(n/2);
            a+2
        }
    }
```

This function was compiled to the following C program using the ternary operator:

```
int32_t rec(int32_t n_0) {
 // Prerequisite
  int32_t a_0 = rec(n_0 / 2);
 // Pure expression
  return (n_0 == 0) ? (1) : (a+2);
}
```

Here the variable a is a prerequisite and is thus evaluated before the return. This makes the if-statement not lazy and creates an infinite recursion causing a $Segfault$.
Then we decided to make use of the usual if-statement of C. Although this statement is not a *pure expression* we can still use it as a prerequisite and pass a newly created variable ite (which will be assigned in each block respectively) in the pure expression. The last example thus becomes:

```
int32_t rec(int32_t n_0) {
 // Prerequisite
  int32_t ite_0;
  if(n_0 == 0){
     ite_0 = 1;
  } else {
     int32_t a_0 = rec(n_0 / 2);
     ite_0 = a + 2;
  }
```

```
  // Pure expression
  return ite_0;
}
```

Here the creation of variable a is done lazily and there is no infinite recursion anymore.

3.3.5 Pattern Matching

Pattern matching is done with nested C if-statements in a similar manner as what was done in Wasm. Yet there are two subtleties to deal with.

The first consists on how to match over ADS when they are void pointers. The trick is to first cast the void pointer into a int32 pointer. This way we can access the first field of the structure which is always an int32 representing the index of the constructor. After checking whether the constructor has the right index we can cast the pointer to the corresponding structure and start checking its other fields.

Consider the following pattern matching:

```
a match {
  case Apple(5) => {...}
}
```

This becomes, assuming the constructor index of Apple is 3:

```
if((*(int32_t*)a_0) == 3 &&          // Test constructor index
  ((struct Apple*)a_0)->att0==5)  // Cast pointer and test field
  {...}
```

The second one concerns $IdPattern$. When an id pattern is encountered in a match case we need a way to initialise its value. This cannot be done in the prerequisites of the pattern matching since it could be a miss match and then the assignment would cause a $SegFault$. Hence, it needs to be assigned uniquely after the match case has succeeded (inside the if-statement block). We then need a way to write code not before but after a *pure expression*! We will call this a *postrequisite* by opposition to a prerequisite. This is implemented in the $matchCond$ function. This function returns three Code types instead of one in Wasm:

1. The Pure Expression (always a boolean expression)
2. The Prerequisite
3. The *Postrequisite*

Consider the following pattern matching:

```
l match {
  case Cons(h,_) => {...} }
```

This becomes, assuming the constructor index of Cons is 1:

```
// Prerequisite (Here none)
  ...
// Pure Expression (to test the case)
 if((*(int32_t*)a_0) == 1 && 1 && 1){
// Postrequisite (assign h_0 to the value)
  int32_t h_0 = ((struct L_Cons*)l)->att0;
  ...
 }
```

4. Possible Extensions

From a purely functional perspective, our code works on any Amy code and can thus be considered as very robust. One notable extension that was not required, but that we did anyways to a certain degree is pretty printing C code. Indeed, generating C code like we presented previously, if done naively, will print all tokens on one line only. Although correct, this is hardly readable for a human.

To solve a part of this problem, we implemented the *mkLines* function in the File Printer object. This function is responsible for adding the appropriate new line characters at the correct places (ie: after a semi-column and braces). Additionally, this function also adds the correct indentation at every line.

One change we did not do, but would definitely improve the readability further is renaming the attributes of a structure into its corresponding Amy name. This cannot be done trivially without changing the underlying architecture, reason we did not do it.

One final extension which doesn't really relate to the C backend but would be very useful nevertheless is developing a garbage collector or an ability to free pointers manually. We indeed have a lot of memory leaks which may hurt sensitive people. Yet, this would require changing Amy first, as Amy doesn't support a native garbage collecting either.