Improved string support

Compiler Construction Final Report

Justin Deschenaux Alexis Roger
justin.deschenaux@epfl.ch, alexis.roger@epfl.ch
Computer Language Processing, CS-320, EPFL

Abstract

In the 1950s the first compilers where developed and these revolutionised programming. Today, they remain a cornerstone tool at a programmers disposal as they allow humans to easily write higher level code which they understand and then compile and distribute it to many different machines, not all using the same instructions. Many take this crucial piece of the puzzle for granted but we decided to peak behind the curtain.

1. Introduction

We build a compiler that would take Amy code as input to generate WebAssembly code. Amy is a purely functional programming language which uses a subset of the Scala language as a model, making the compiler simpler to implement. On the other hand, WebAssembly is a portable binary code format which can be easily executed in a browser or in a nodeJS environment. In order to output code, the compiler processes programs in several steps: lexing, parsing, name analysis, type checking and finally code generation.

Here is an explanation of these different layers. In the **lexer** phase, we use a Scala extension named *Scallion* in order to run thought the code using regular expressions and to 'tokenize' it. This list of tokens is then fed to the **parser** who will build a syntax tree, abstracting away details like parentheses or separators. In order to keep the parsing time linear, we implemented a restrictive LL(1) grammar, as required by *Scallion*. The AST is then passed onto the **name analyzer**, whose duty is to make sure that all function and variable references are well defined, in addition to assigning unique identifier to them in case of correctness. The penultimate stage is the **type checker**, which ensures that the expected types match the actual ones. Finally, the **code generator** can export our valid program into an (ideally!) as correct WebAssembly one.

Since there is only few operations allowed on String in Amy, we chose to improve the current implementation with slicing a Char type, and derived functionalities . In order to adapt Amy specification, we had to modify each of the above layers. However, the **code generation** one was modified the most.

You may find a link to our GitHub repository in the references for our complete code.

2. String Length

The initial modification that we made was to store the length of the string before said string. In the original implementation, strings are address to the first character of the sequence. We added a simple function able to return the length of the string in constant time, by putting the value before the first character in memory. However, the String referenced by variables points on the first character in order lower the amount of changes in existing code (especially for printing). Hence, when accessing the length, we only request the 32-bits integer stored 4 bytes earlier, with *"[string pointer]-4"*. Amy having only signed integers, this method can only store lengths up to $2^{31} - 1$ before overflowing. We considered this and then realised that this would mean that our string is more than 2 billion characters, taking up more than 2 gigabytes of memory. Furthermore, the memory is relatively small and considering the fact that we do not do any garbage collection, the memory risks being saturated even before we get a String close to this size. This is an additional argument for why we disregarded the overflow case.

In the unlikely case were this length limit is passed and we have enough memory, we need our string to be more that $2^{32} - 1$, more that 4 billion characters so 4

gigabytes, to create a possible memory leak. Between 2 billion and 4 billion characters, the length would be negative, meaning the program execution becomes undefined.

3. The character type

The first change that we did was to implement a new primitive type to the Amy language. This type is the character type, named "Char" in the code and stores a single ASCII character. This can be though of as a string of length 1 but takes only half of the space. A character is stored on the heap in the same way as a 32 bit integer. This means that there is no referencing issue as the character is directly stored on the heap in its' ASCII value.

A string of 1 character would take 3 * 32 bits of memory:

1. The string reference pointer
2. 1 byte for character, 3 bytes of padding
3. The length value

On the contrary, a character only takes up 32 bit on the heap, 1 byte to store the ASCII value and 3 bytes of padding.

We chose to accept special values for characters, such as "\n". They are stored as their value "[return]" and not as their representation (like for Strings, where "\n" stands for 2 characters). Originally, we wanted to also accept them in String literals, but time constraints forced us to focus on other parts of the compiler. This however brings the drawback that the only way to have "\n" in text is to concatenate it with a character somehow.

Here is an example of what characters look like:

```
object charDemo {
  // 1 character Char literal
  val c0: Char = 'A';
  // Special character Char literal
  val c: Char = '\n';

  // String with return character
  val s: String = "Very special string" ++
                  Std.charToString(c);
}
```

4. String slicing

The biggest task, and the one on which we spent the most time, is string slicing. Our goal was to implement a string slicing with the easiest and most pleasant usage by the user. To this end, we did our best to make it as flexible as possible, never failing (except when provided with a step value of 0) and covering all the corner cases. The first step was to recognize a slicing expression in the parser. But as slices may be daisy chained, we needed to recognise the following regular expression:

$$simpleExpr([expr? : expr?(: expr?)?])^*$$

Where the ".*" is the Kleene operator and "?" denotes an optional parameter. We only allowed slicing on simple expressions (literal, variables, function calls and constructors). On the other hand, the parameters being expressions, they may be anything, from integer to pattern matching or even contain an other slicing. In the type checker we will nevertheless verify that the leftmost is a string and that the three in the square brackets have integer type.

We will now refer to the slicing operation with the following nomenclature: $string[start : end : step]$ and we will focus on the code generation step.

At our first attempt, we only accepted positive $start$ and end (we crashed the program with a not so useful error message, when parameters where illegal) and replaced a missing parameter by a default one at compile time. For the $step$ parameter, the default value was (and is still) 1. For the $start$ and end parameters, it is a little less simple. In the case where the step was positive, the default parameter for $start$ was 0 and for end, $length(string)$. If however the step provided was negative, $start$ would take the value $length(string) - 1$ and end 0. The issue was that since the end argument is excluded from the slicing (while $start$ is included), there would be no way to keep the first character while slicing backwards... An idea was to set end to -1, but then it would have failed the run-time tests. The issue were consequences of the fact that half the information was available at compile time (whether parameters where all provided) and half at runtime (if the step was negative or positive).

After facing those problems, we decided to choose a different approach for the implementation, allowing at the same time negative indexing (negative $start$ and end, more about that in the example section). Upon call, the slicing needs more parameters in order to check if $start$ and end where provided or not. We also only use end as a temporary value, used for computing

the length of the sliced String. Instead of explaining everything with sentences, we think that some pseudocode is clearer in order to understand how the implementation works.

```
// The clip function is auxiliary and is used in order to
// bound parameters in slicing and character addressing
// (charAt function)
def clip(value, max): = {
  if(value < 0) {
    value += length
  }
    value = max(0, value)
    min(length, value)
}

// This is how the parameters are computed and bounded.
// Note that this functionality is implemented in WASM
if step > 0: // step < 0
  // Compute start value
  if start is default:
    start = 0 // start = length − 1
  else:
      start = clip(start, length(str))
      // start = clip(start, length(str) − 1)
   // Compute range, sliced string length if step is 1
   if end is default:
      range = length(str) − start
        // range = start + 1
   else:
      end = clip(end, length(str))
      range = max(end − start, 0)
         // range = max(0, start − end)
```

Here under you may find a series of slicing demonstrations, all with their desired outcome commented out next to them. These cover all the corner cases and demonstrate the key features.

```
object slicingDemos {
// Basic slicing. For readability, used literals
"asdf"[::−1]; // fdsa
val str: String = "aabbccddee";
str[::4]; // ace
str[::−4]; // eca
str[−3: 2: −2]; // dcb
str[:1000:4]; // ace

// Slicing with expressions
val e0: Char = 'A';
str[e0 match {
case 'E' => 0
case 'A' => 999
}:: −1]; // eeddccbbaa
// Similar test can be conducted with conditional expression
// Chained slicing
str[0:5][::−1][2:99]; // baa
}
```

5. String comparison

A simple yet essential modification that we made was to allow string comparison by value. Given that we implemented a $charAt$ function, implementing $equals$ was straightforward (and done in pure Amy). This function is detailed bellow in **additional functions**.

Following up is a demonstration of string comparisons. It should be noted that before all of these examples would have returned $false$.

```
object demo {
val str0: String = "aaa";
val str1: String = "abc";

S.equals(str0, str1); // False
S.equals(str0, "aaa"); // True
S.equals(str1, "cba"[::−1]); // True
}
```

6. Additional functions

Here under you may find a list of additional functions that were implemented along with examples of their usage.

- $charAt(s : String, i : Int) : Char$

This function return the character at the nth position in the string s. First we wanted to implement it as a slicing with a single expression argument. The issue is that the regular expression for slicing and $charAt$ are intricate and we didn't manage to find a clean way to handle it in reasonable time.

- $replace(s : String, toReplace : String, pattern : String) : String$

Goes through the string s and replaces every occurrence of the sub-string $toReplace$ with the string $pattern$

- $strip(s : String) : String$

Removes all the blank characters (newlines, tabs, spaces...) from the start and the end of the string s.

- $caesarCipher(plainText : String, offset : Int, upper : Boolean) : String$

A nice example of the upgraded string implementation is this Caesar cipher. We need to pass it a string to crypt, the amount by which to shift the letters (may

be negative) and weather or not we are working with capitals.

- $caesarDecipher(cipherText : String, offset : Int, upper : Boolean) : String$

Analogous to the one above.

```
object demo{
  val str: String = "aabb";
  S.charAt(str, 1); // a
  S.charAt(str, -1); // b

  S.replace("This That This That This That", "That", "a");
    // This a This a This a

  S.replace("This That This That This That", "That",
"Tuhuauauaut"[::2]);
    // This Thaaat This Thaaat This Thaaat



  val str3: String = "  aabb ";
  S.strip(str); // aabb

  val plainText: String = "ABCDEFA";
  val cipherText: String =
    S.caesarCipher(plainText, 66, true); // OPQRSTO
  val deciphered: String =
    S.caesarDecipher(cipherText, 66, true) // ABCDEFA
}
```

7. Possible Extensions

We were able to finish most of what we had planned to do and had presented in December. $Drop$ and $take$ are 2 functions that we chose not to implement and instead implement the default parameters in the slicing. A $drop(string, n)$ is the same as $string[n :]$ and $take(string, n)$ is the same as $string[: n]$. Furthermore, we did not implement either a split function, nor a count function, nor a contains function.

An additional functionality that would have been a nice addition would have been string interpolation. In other words, being able to evaluate expressions inside of strings. For instance, "$I\ am\ \${"20\ years\ old"[: 2]}$" would evaluate to "$I\ am\ 20$". This simplifies greatly the printing of strings that are not straight forward and that contain the value of one or multiple operations.

8. References

Specification of the Amy language
http://lara.epfl.ch/~gschmid/clp19/amy-spec.pdf
Autumn 2019

Project Github
https://github.com/Dudier/CLP
January 2020