COMS W3998

String Literals in SSLANG

Anjali Smith

12/23/22

Abstract

SSLANG (Sparse Synchronous Language) is a functional programming language built on top of the Sparse Synchronous Model, a runtime system proposed by Professor Stephen Edwards and John Hui that allows precise timing control. This semester, I worked with the language design team, led by Emily Sillars, to add Haskell style list literal syntax to SSLANG. This work required an understanding of existing SSLANG syntax, the SSLANG codebase, compiler architecture, and representations of string literals in the different phases of the compiler.

1 Motivating Example:

The following SSLANG program prints out the string ”Hello” followed by a newline character. The program works by inserting the ascii value of each character in the word ”Hello”, with the last two characters being a newline character and null terminator, into the standard output stream.

```
main (cint : &Int) ( cout : & Int ) -> () =
  after 10 , (cout : & Int) <- 72
  wait (cout : & Int)
  after 10 , (cout : & Int) <- 101
  wait (cout : & Int)
  after 10 , (cout : & Int) <- 108
  wait (cout : & Int)
  after 10 , (cout : & Int) <- 108
  wait (cout : & Int)
  after 10 , (cout : & Int) <- 111
  wait (cout : & Int)
  after 10 , (cout : & Int) <- 10
  wait (cout : & Int)
  after 10 , (cout : & Int) <- 0
  wait (cout : & Int)
```

To make SSLANG more user-friendly, we wanted to add Haskell style string literal syntax so that the following program could print ”Hello” in the same manner as the previous program:

```
type String
  Cons Int String
  Nil

putc (cout : &Int) c =
  after 1, cout <- c
  wait cout

puts cout s =
  match s
    Cons c ss =
      after 1, cout <- c
      wait cout
      puts cout ss
    Nil = ()

main cin cout =
    let s = "Hello"
    puts cout s
    putc cout 10
```

In the above program, the user defined a String type, a function that prints out a string ("puts"), a function that prints out a character("putc"), and combined all of them to print out the string "Hello" followed by a newline.

2 Attempted Implementation - Lowering Stage

In the SSLANG codebase, there was already support for string literals in the scanner and parser which was generating an expression node of type LitString in the Abstract Syntax Tree(AST). However, once the lowering stage is reached and AST nodes are transformed into equivalent Intermediate Representation(IR) nodes, there was no code to transform string literal nodes into their appropriate IR nodes. Therefore, when we attempted to run a program with our desired list syntax, our program would throw a compiler error and halt the compiler pipeline during the lowering phase.
My initial plan to add string literal syntax to SSLANG was to transform AST nodes of type LitString into its equivalent IR node by adding a case for LitStrings when lowering AST expression nodes into IR expression nodes. More specifically, I needed to transform a Lit instance of an AST Expression node into an App instance of an IR Expression node.

2.1 Lowering AST Expression Nodes into IR Expression Nodes

The following code shows the AST Expr data type and the Literal data type from the file "AST.hs":

```
data Expr
  = Id Identifier
  | Lit Literal
  | Apply Expr Expr
  | Lambda [Pat] Expr
  | OpRegion Expr OpRegion
  | NoExpr
  | Let [Definition] Expr
  | While Expr Expr
  | Loop Expr
  | Par [Expr]
  | IfElse Expr Expr Expr
  | After Expr Expr Expr
  | Assign Expr Expr
  | Constraint Expr TypAnn
  | Wait [Expr]
  | Seq Expr Expr
  | Break
  | Match Expr [(Pat, Expr)]
  | CQuote String
  | CCall Identifier [Expr]
  | Tuple [Expr]
  | ListExpr [Expr]
  deriving (Eq, Show, Typeable, Data)

data Literal
  = LitInt Integer
  | LitString String
  | LitRat Rational
  | LitChar Char
  | LitEvent
  deriving (Eq, Show, Typeable, Data)
```

The following code shows the Expr data type for an expression node in the IR from the file ”IR.hs”:

```
data Expr t
  = Var VarId t
  | Data DConId t
  | Lit Literal t
  | App (Expr t) (Expr t) t
  | Let [(Binder, Expr t)] (Expr t) t
  | Lambda Binder (Expr t) t
  | Match (Expr t) [(Alt, Expr t)] t
  | Prim Primitive [Expr t] t
  deriving (Eq, Show, Typeable, Data,
```

In order to transform an AST LitString to an IR App node, I added a case to the following function in the file ”LowerAST.hs”:

```
lowerExpr :: A.Expr -> Compiler.Pass (I.Expr I.Annotations)
```

When a LitString is passed to lowerExpr, it has the type:

```
Lit (LitString s)
```

This type needs to be transformed into the following during the lowering stage:

```
App (Expr t) (Expr t) t
```

where t is the type annotation for each IR node.

2.2 New Case in LowerExpr

In order to lower the LitString, I added the following case to the lowerExpr function and a helper function called buildAppList:

```
-- New case iin lowerExpr:
lowerExpr (A.Lit (A.LitString (h:t))) = return $ I.foldApp (I.App (I.Data "Cons" ty

-- Helper function for new case:
buildAppList :: A.Expr -> [(I.Expr I.Annotations, I.Annotations)]
buildAppList (A.Lit (A.LitString "")) = [I.App (I.Data "Cons"  typ) (I.Data "Nil" t
```

2.3 Sample Lowering of LitString

To demonstrate how an AST LitString is transformed into an IR App, I will walk through an example. In the following program, assume that the type String and the function "puts" have both been defined already,

```
{-
-- Hidden implementation of puts and String type
-}
main cin cout =
     let s = "abc"
     puts cout s
```

If "abc" was represented by an AST expression node like:

```
Lit (LitString "abc")
```

Its corresponding representation in the IR would be a nested application of data constructors:

```
(App ( App (Id "Cons" t) (lit 'a' t) t) (App (App (Id "Cons" t) (Lit 'b' t) t) (App
t = (Annotation [])
```

3 Improved implementation - Code Reuse

During a weekly research meeting, after updating the group about my progress with the previously discussed approach to add string literal syntax to SSLANG, we realized that the work for lowering a listExpr and a LitString in the lowerExpr function was almost identical. In order to reuse code, we decided to have string literals enter the lowering phase as a desugared ListExpr, representing a list of ascii characters, so that both listExprs and litStrings used the same case in the lowering stage.
This change was implemented by having two sub-stages in the front-end compiler stage: one to desugar LitStrings and one to desugar ListExprs. The desugarString stage transforms AST nodes of type LitString into AST nodes of type ListExpr. The desugarList stage, which Max worked on, transforms AST nodes of type ListExpr into AST nodes of type App, which will eventually be lowered to IR nodes of type App in the lowering stage.

3.1 DesugarStrings.hs

In the file DesugarStrings.hs, I implemented the function desugarExpr which performs the conversion from an AST node of type LitString to an AST node of type ListExpr. One mistake I had made originally was only casing on an input to the function of the form: Lit (LitString s), when in reality, string literals can be wrapped within many other AST node types such as the types While, Loop, Apply, and the numerous other cases listed in my code:

```
desugarExpr :: Expr -> Expr
desugarExpr (Lit (LitString s)) = ListExpr (convertList s)
desugarExpr (Apply e1 e2) = Apply (desugarExpr e1) (desugarExpr e2)
desugarExpr (Lambda p e) = Lambda p (desugarExpr e)
desugarExpr (OpRegion e o1) = OpRegion (desugarExpr e) o1
desugarExpr (Let d e) = Let (map extractExpr d) (desugarExpr e)
desugarExpr (While e1 e2) = While  (desugarExpr e1) (desugarExpr e2)
desugarExpr (Loop e) = Loop (desugarExpr e)
desugarExpr (Par ls) = Par (map desugarExpr ls)
desugarExpr (IfElse e1 e2 e3) = IfElse (desugarExpr e1) (desugarExpr e2) (desugarEx
desugarExpr (After e1 e2 e3) = After (desugarExpr e1) (desugarExpr e2) (desugarExpr
desugarExpr (Assign e1 e2) = Assign (desugarExpr e1) (desugarExpr e2)
desugarExpr (Constraint e t) = Constraint (desugarExpr e) t
desugarExpr (Wait ls) = Wait (map desugarExpr ls)
desugarExpr (Seq e1 e2) = Seq (desugarExpr e1) (desugarExpr e2)
desugarExpr (Match e1 [(p, e2)]) = Match (desugarExpr e1) [(p, desugarExpr e2)]
desugarExpr (CCall i ls) = CCall i (map desugarExpr ls)
desugarExpr (ListExpr es) = func es
desugarExpr e = e

func :: [Expr] -> Expr
func [] = Id (Identifier "Nil")
func (h:t) = Apply (Apply (Id (Identifier "Cons")) h) (func t)

convertList :: [Char] -> [Expr]
convertList ls = [Lit (LitInt (toInteger i)) | i <- map ord ls]

extractExpr :: Definition -> Definition
extractExpr (DefFn i p t e) = DefFn i p t (desugarExpr e)
extractExpr (DefPat p e) = DefPat p (desugarExpr e)
```

3.2 Simplification with "Scrap your Boilerplate" Approach

In order to simplify our passes, Emily used the "Scrap your Boilerplate" approach (SYB) which is a programming approach that allows users to integrate generic programming in Haskell. After Emily made the AST derive Typeability and Data, I was able to use the function "everywhere" to desugar nodes of type LitString that are wrapped by nodes of other types without casing on every possible type in desugarExpr as I had done before. Using the SYB approach, I added calls to "everywhere" and removed the additional cases in desugarExpr. The following code is my simplified version of the functions desugarStrings and desugarExpr in the file desugarStrings.hs:

```
-- | Desugar String Literal nodes inside of an AST 'Program'.
desugarStrings :: Program -> Compiler.Pass Program
desugarStrings (Program decls) = return $ Program $ desugarTop <$> decls
 where
  desugarTop (TopDef d) = TopDef $ desugarDef d
  desugarTop t          = t

  desugarDef (DefFn v bs t e) = DefFn v bs t $ everywhere (mkT desugarExpr) e
  desugarDef (DefPat b e    ) = DefPat b $ everywhere (mkT desugarExpr) e

-- | Transform a node of type LitString into a node of type ListExpr
-- For ex, (Lit (LitString "abc")) turns into (ListExpr [97, 98, 99])
desugarExpr :: Expr -> Expr
desugarExpr (Lit (LitString s)) = ListExpr (convertList s)
  where convertList ls = [ Lit (LitInt (toInteger i)) | i <- map ord ls ]
desugarExpr e = e
```

3.3 Test cases

In order to test the string literal syntax, I added two test cases: one that prints "Hello" followed by a newline character and one that prints an empty string.

3.3.1 Printing "Hello"

```
type String
  Cons Int String
  Nil

putc (cout : &Int) c =
  after 1, cout <- c
  wait cout

puts cout s =
  match s
    Cons c ss =
      after 1, cout <- c
      wait cout
      puts cout ss
    Nil = ()

main cin cout =
    let s = "Hello"
    puts cout s
    putc cout 10
```

3.3.2 Empty String Test

```
type String
  Cons Int String
  Nil

puts cout s =
  match s
    Cons c ss =
      after 1, cout <- c
      wait cout
      puts cout ss
    Nil = ()

main cin cout =
    let s = ""
    puts cout s
```

4 Final outcome

After adding support for string literal syntax in SSLANG and two test cases, I squashed and merged my commits in to the main branch. In the future, after implementing string library functions, it would be nice to test the desugaring proccess of string literals wrapped within Expr nodes of other types like IfElse and While since we are currently only testing string literal syntax within a let expression.

5 Next Steps

Next semester, I will be working on implementing two functions from the OCaml list library functions, specifically cons and append, and the append to front list operator.

References

1. Ralf Lämmel and Simon Peyton Jones. 2003. Scrap your boilerplate: a practical design pattern for generic programming. SIGPLAN Not. 38, 3 (March 2003), 26–37. https://doi.org/10.1145/640136.604179
2. Stephen A. Edwards and John Hui. The Sparse Synchronous Model. In Forum on Specification and Design Languages (FDL), Kiel, Germany, September 2020.