COMS 6901 E

SSLANG Algebraic Data Types and Beyond

Emily Sillars

5/12/22

Abstract

My prior report concluded with C code designs for algebraic data types and the memory manager fully solidified, and news of an updated runtime system merged into the main codebase. Now that a C code representation of ADTs has been established, what further steps must we take before we can translate a simple SSLANG source program with ADTs into an executable? After building this foundation, how can we extend ADT functionality? Where do we go from here? This research report is split into two main sections, firstly, full integration of ADTs into and further work on the compiler pipeline, and secondly, example SSLANG programs and a look into the future.

1 Integrating ADTs + Pretty Printing

1.1 Updating Codegen.hs

The updated runtime system included a number of helpful C macros and functions in ssm.h for common ADT related tasks, like allocating space for ADTs and accessing their fields. In addition, the new LibSSM.hs file defined useful, reusable haskell functions for generating the c quasiquotation expressions for calling these macros. Together, ssm.h and LibSSM.hs provided another layer of abstraction between codegen.hs and calls to the language-c-quote library.

A helpful function and macro from ssm.h:

```
/** Allocate a new ADT object on the heap. */
ssm_value_t ssm_new_adt(uint8_t val_count, uint8_t tag);

/** Access the field of an ADT object. */
 ssm_adt_field(v, i)                                                    \
  (&*container_of((v).heap_ptr, struct ssm_adt1, mm)->fields)i
```

Corresponding functions in LibSSM.hs, which call the C quasiquotation library:

```
-- | @ssm_new_adt@, allocate a new ADT object on the heap.
new_adt :: Int -> DConId  -> C.Exp
new_adt val_count tag = cexp|ssm_new_adt($uint:val_count, $id:tag)|

-- | @ssm_adt_field@, access the @i@th field of an ADT object. Assignable.
adt_field :: C.Exp -> Int -> C.Exp
adt_field v i = cexp|ssm_adt_field($exp:v, $uint:i)|
```

One of the first tasks at hand was to update codegen.hs to make use of these in functions Lib.SSM wherever possible. Here is an example of a change I made to codegen.hs to use the updated runtime system. In the following code snippets, the code for allocating and initializing an ADT is cleaned up.

Snippet from old codegen.hs, hard coding ssm.h identifiers:

```
let alloc = citem|$exp:tmp = $id:ssm_new($int:sz,$id:tg);|
let initField = ( \y i ->
                  citem| $exp:tmp->$id:payload$uint:i = $exp:y;|
                )
let initFields = zipWith initField argVals 0 :: Int, 1 ..
```

Snippet from updated codegen.hs, making use of LibSSM.hs:

```
let alloc      = citem|$exp:tmp = $exp:(new_adt sz tg);|
let initField  = \y i -> citem|$exp:(adt_field tmp i) = $exp:y;|
let initFields = zipWith initField argVals 0 :: Int, 1 ..
```

While the changes to codegen.hs themselves were relatively mechanical, this newly added layer of abstraction decreased code duplication and made the code generation process more modular overall.

1.2 Updating TypeChecker.hs, and HM.hs: colors.ssl

After updating ADT code generation to work with the updated runtime system, I moved to the middle of the compiler pipeline to update SSLANG's typechecker and type inference algorithm to work with ADTs. Most of the work at this stage included augmenting the type inference state to include information from previously parsed type definitions. Let's consider an example SSLANG program, colors.ssl, to make this idea more concrete.

```
type Color
  White
  Black
  RGB Int Int Int

println ( cout : & Int ) ( n : Int ) -> () =
  // omitted for brevity; see appendix
printColor ( cout : & Int ) (c : Color) -> () =
  // omitted for brevity; see appendix

main ( cout : & Int ) -> () =
  let licorice = Black
  let lemon = RGB 255 255 0
  let lime = RGB 161 242 0
  printColor cout lime
  printColor cout lemon
  printColor cout licorice
```

To type check this program, one needs to know that the arguments to data constructor RGB should each be of type Int, and that an instance of RGB or Black is an instance of type Color. This

type information comes from the ADT type definition at the top of the program, and needs to be accessible while the type checker traverses main's abstract syntax tree.

For this reason, I augmented the type inference monad state, inferstate, with two maps, dConType and dConArgType, which mapped data constructors to their type constructor and data constructors to their list of argument types respectively:

```
-- | Inference State.
data InferState = InferState
  { varMap        :: M.Map I.VarId Classes.Scheme
  , equations     :: (Classes.Type, Classes.Type)
  , unionFindTree :: M.Map Classes.Type Classes.Type
  , count         :: Int
  , dConType      :: M.Map I.DConId Classes.Scheme
  , dConArgType   :: M.Map I.DConId Classes.Type
  }
```

I also made sure to fill in these maps with relevant information from the program node's type definitions before type inference took place:

```
-- | 'inferProgram' @p@ infers the type of all the
      programDefs of the given program @p@.
inferProgram :: I.Program Ann.Type -> Compiler.Pass (I.Program Classes.Type)
inferProgram p = runInferFn $ do
  typeDefs' <- recordADTs $ I.typeDefs p
  recordFDefs $ I.programDefs p
  defs'     <- inferProgramDefs $ I.programDefs p -- occurs after recordADTs
  return $ I.Program { I.programDefs  = defs'
                     , I.programEntry = I.programEntry p
                     , I.typeDefs     = typeDefs'
                     }
```

While implementing recordADTs, I ran into some type checking errors and asked Xijiao Li for feedback and guidance. She helped update HM.hs to typecheck ADTs correctly, pointing out that the type of a data constructor is only its type constructor when that data constructor is nullary! The rest of the time (when a data constructor possesses arguments), the type of a data constructor is actually a function that returns an ADT. Xijiao added the function buildDConType (a helper for recordADTs) to the HM.hs file in my branch which highlights this distinction and fixed the final bug in ADT type inference. A call to recordADTs will process the Color type definition,

```
type Color
  White
  Black
  RGB Int Int Int
```

and fill in the the dConType and dConArg type like so:

```
// dConType :: M.Map I.DConId Classes.Scheme
(Black, Color)
(White, Color)
(RGB, Int -> Int -> Int -> Color)
```

```
// dConArgType :: M.Map I.DConId [Classes.Type]
(Black, [])
(White, [])
(RGB, [Int, Int, Int])
```

Since SSLANG is a functional language, treating a data constructor like a function at the type inference stage makes sense. In fact, to further extend the functionality of ADTs, we needed to "treat data constructors like functions" at the code generation step as well.

1.3 Partially Applied Data Constructors: mixingColors.ssl

The addition of closure code generation to the main codebase in early April provided support for partial application of functions in SSLANG. However, due to a distinction made between functions and data constructors at the code generation level, partial application of data constructors in SSLANG still wasn't possible. Now that the simplest of ADT programs worked successfully end-to-end, our next goal was to support partial application of data constructors. This could be easily attained by wrapping fully applied data constructors inside a function, essentially creating a top-level "constructor function" for each data constructor. Let's use another example, mixingColors.ssl, to make our discussion more concrete.

```
type Color
  White
  Black
  RGB Int Int Int

println ( cout : & Int ) ( n : Int ) -> () =
  // omitted for brevity; see appendix

printColor ( cout : & Int ) (c : Color) -> () =
  // omitted for brevity; see appendix

main ( cout : & Int ) -> () =
  let print = printColor cout
  let lime = RGB 161 242 0
  let base = RGB 255 147
  print (base 0)          // orange
  print (base 255)        // rose
  print (base 142)        // peach
  print lime
```

With our current code generation schema, fully applied data constructors like lime do indeed have a corresponding C code translation.

```
let lime = RGB 161 242 0
// corresponds to a block of C code statements
acts->__tmp_0 = ssm_new_adt(3U, RGB);
ssm_adt_field(acts->__tmp_0, 0U) = ssm_marshal(161);
ssm_adt_field(acts->__tmp_0, 1U) = ssm_marshal(242);
ssm_adt_field(acts->__tmp_0, 2U) = ssm_marshal(0);
acts->lime = acts->__tmp_0;
```

Partially applied data constructors, in contrast, do not.

```
let base = RGB 255 147
// corresponds to ???
```

Instead of directly translating lime to block of C statements, consider wrapping code for a fully applied data constructor inside a top level constructor function:

```
__RGB (__arg0 : Int) (__arg1 : Int) (__arg2 : Int) -> Color =
  RGB __arg0 __arg1 __arg2
```

Which then corresponds to a C function:

```
void __step___RGB(ssm_act_t *actg)
{
    act___RGB_t *acts = container_of(actg, act___RGB_t, act);

    switch (actg->pc) {

      case 0:
        ;
        acts->__tmp_0 = ssm_new_adt(3U, RGB);
        ssm_adt_field(acts->__tmp_0, 0U) = acts->__arg0;
        ssm_adt_field(acts->__tmp_0, 1U) = acts->__arg1;
        ssm_adt_field(acts->__tmp_0, 2U) = acts->__arg2;

      default:
        break;
    }
    *acts->__return_val = acts->__tmp_0;

  __leave_step:
    ssm_leave(actg, sizeof(act___RGB_t));
}
```

Now fully applied data constructors simply translate to a fully applied function. By extension, partially applied data constructors translate to a partially applied function. Once data constructors become functions, they can make use of SSLANG's closure representation and get partial application for free!

```
main ( cout : & Int ) -> () =
  let print = printColor cout
  let lime = __RGB 161 242 0
  let base = __RGB 255 147
  ...
```

In PR 85, I added an IR to IR transformation pass to the compiler called DConToFunc.hs, which created constructor functions for each user defined type and replaced all data constructor applications with calls to that data constructor's corresponding constructor function.

Before DConToFunc.hs pass:

```
main ( cout : & Int ) -> () =
  let print = printColor cout   // :: Color -> ()
  let lime = RGB 161 242 0      // :: Color
  let base = RGB 255 147        // :: Int -> Color
```

After DConToFunc.hs pass:

```
__RGB (__arg0 : Int) (__arg1 : Int) (__arg2 : Int) -> Color =
  RGB __arg0 __arg1 __arg2
main ( cout : & Int ) -> () =
  let print = printColor cout     // :: Color -> ()
  let lime = __RGB 161 242 0      // :: Color
  let base = __RGB 255 147        // :: Int -> Color
  ...
```

In my first iteration of dConToFunc.hs, I implemented an optimization which excluded fully applied data constructors from this transformation process. The idea here was that fully applied data constructors don't actually need a closure object, so we could save heap space by directly translating these without a corresponding function call. During a code review, Professor Stephen Edwards advised me to remove this optimization from dConToFunc.hs for simplicity, and to save this optimization for a later pass in the compiler. Removing this optimization greatly simplified my code.

Since the dConToFunc pass runs after type inference, the task of generating IR level constructor functions included type annotating them correctly. Manually coding the types of these constructor functions emphasized for me the importance of viewing data constructors as functions.

```
RGB             :: Int -> ( Int -> ( Int -> Color ) )
RGB 255         :: Int -> ( Int -> Color )
RGB 255 147     :: Int -> Color
RGB 255 147 142 :: Color
```

Having previously worked on the code generation side of the compiler all last semester, I'd spent the majority of my time viewing data constructors as a single, fully applied unit. I realize now that while a data constructor and its arguments can still be viewed this way, it can also be viewed as a nested application expression, with each nested sub-expression containing a different type. This latter view helps me understand how algebraic data types are less extensions of and more manifest pieces in the world of functional languages.

1.4 Pretty Printing the IR

An important debugging tool for any compiler under development is its pretty printer. After completing partial application of data constructors, I worked on issue 51, "Prettify pretty printer", aiming to make the output of the IR pretty printer look like readable user-written source code. The current pretty printer output, nicknamed "spaghetti" implemented the pretty typeclass from the haskell module prettyprinter over each node of SSLANG's intermediate representation. Here is the spaghetti output for an abbreviated version of colors.ssl:

```
(__RGB,(fun __arg0
{(fun __arg1 {(fun __arg2 {((((RGB: (((Int32 -> Int32) -> Int32) -> (Color)))
(__arg0: Int32): (Int32 -> (Color)))
```

```
(__arg1: Int32): ((Int32 -> Int32) -> (Color)))
(__arg2: Int32): (Color))}: (Int32 -> (Color)))}:
(Int32 -> (Int32 -> (Color))))}: (Int32 -> (Int32 -> (Int32 -> (Color))))))
(main,(fun cout
{((let {lime = {((((__RGB: (Int32 -> (Int32 -> (Int32 -> (Color)))))
(161: Int32): (Int32 -> (Int32 -> (Color)))) (242: Int32): (Int32 -> (Color)))
(0: Int32): (Color))}};
((let {base = {(((__RGB: (Int32 -> (Int32 -> (Int32 -> (Color)))))
(255: Int32): (Int32 -> (Int32 -> (Color)))) (147: Int32):
(Int32 -> (Color)))}}; ((): ())): ())): ())}: ((&Int32) -> ())))
```

While this output is difficult to read, it represents an accurate, one-to-one representation of our IR. I'd previously made use of this detailed spaghetti output when checking the correctness of my dConToFunc pass, especially the typing of the generated constructor functions. This experience made me hesitant to throw away potentially useful information for the sake of clarity. In an attempt to preserve the original pretty printer output, I built a typeclass called format on top of pretty, and added a flag to turn type annotations on and off. The format typeclass would call pretty to get the barebones representation of a piece of the IR, then wrap it in whitespace and indentations accordingly. To complicate matters, format also performed some IR simplifications like un-nesting function application, which caused it to completely replace a call to pretty for certain IR nodes.

During a code review, John Hui and I discussed the best way to organize the revised pretty printer code. Acknowledging both the value of the detailed spaghetti output and the need for a clearer pretty printed output, we decided to preserve spaghetti in a separate typeclass, and make the format typeclass the default pretty printer output (take the implementation of format and put it in pretty, and put the old pretty implementation in a separate typeclass). We decided to remove type annotations from the default pretty printer output for readability and simplicity; a pretty printer with more fine tuned/selective type annotations will be left for a future PR. The current pretty printed output for the abbreviated colors.ssl on my pretty-print branch appears as follows:

```
type Color
  White
  Black
  RGB Int Int Int

__RGB (__arg0 : Int) (__arg1 : Int) (__arg2 : Int) -> Color =
  RGB __arg0 __arg1 __arg2

main (cout : &Int) -> () =
  let lime = __RGB 161 242 0
  let base = __RGB 255 147
  ()
```

It's significantly more readable than spaghetti, but there's still much room to grow! In particular, the option for selective type annotations mentioned earlier, and minimal parentheses (currently pretty printed output is correct, but may include unnecessary parentheses) could be added. Finally, I updated our regression tests script runtests.sh to test the pretty printer. For each test program, the script checks

1. Can the compiler produce pretty printed IR?

2. Can the compiler read in this pretty printed IR as source and produce the same output IR?

3. Can the compiler read in this pretty printed IR and produce an executable that behaves the same as the original compiled source program?

All regression tests are passing, and barring some few final tweaks, my pretty-print PR is ready to merge into the main codebase. I expect to make the final changes and merge this PR in the next couple days.

2 Example SSLANG Programs

Near the end of the semester, members of the SSLANG research group participated in a mini hackathon, writing example SSLANG programs on a slightly updated branch off the main codebase called platform-builds.

2.1 A Big Number Library

I implemented the start of a big number library(a library for numbers unbounded in size) in SSLANG. The type Number represents a natural number of any size in decimal format, and subsequent functions define operations on the Number data type. A Number is a list of Digits.

```
type Digit
    Zero
    One
    Two
    Three
    Four
    Five
    Six
    Seven
    Eight
    Nine

type Number
    Nums Number Number
    Dig Digit
```

Notable functions are the itoNum function, which takes a regular SSLANG integer and converts it to a big number, and printNum which prints a big number to the screen.

```
// some helper functions omitted for brevity; see appendix

itoNum (i : Int) -> Number =
    match (i > 10)
        0 = Dig (itod i)
        _ = let r = i % 10
            let q = i / 10
            Nums (itoNum q) (Dig (itod r))
```

```
printNum (cout : & Int) (n : Number) -> () =
    match n
      Dig dig = print cout (dtoc dig)
      Nums d tl = printNum cout d
                  printNum cout tl
```

I also created a bool type to return the result of comparison between two big numbers.

```
type Bool
    True
    False

// eqDigit omitted for brevity; see appendix

eqNumber (n : Number) (n2 : Number) -> Bool =
    match n
      Dig d = match n2
                Dig d2 = eqDigit d d2
                _ = False
      Nums d tl = match n2
                    Nums d2 tl2 = match (eqNumber d d2)
                                    True = match (eqNumber tl tl2)
                                             True = True
                                             _ = False
                                    _ = False
                    _ = False
```

2.2 Animated Wheel

While arguably trivial, wheel3.ssl is by far my favorite SSLANG program. It animates a rotating line in the console window, simulating a spinning "wheel".

```
print ( cout : & Int ) (n : Int) -> () =
  let clear = 13  // '\r'
  after 10, cout <- clear
  wait cout
  after 100000000 , cout <- n
  wait cout

main (cint : &Int) ( cout : & Int ) -> () =
  let left = 92    // '\'
  let vert = 124   // '|'
  let right = 47   // '/'
  let horz = 45    // '-'
  loop
    print cout left
    print cout vert
    print cout right
    print cout horz
```

I'm impressed by SSLANG's ability to express framerate without the need for an explicit counter variable. While the current wheel3.ssl program is not framerate independent, I imagine that delta time, a concept crucial for game programming, could be expressed elegantly using SSLANG's runtime value `ssm_now`.

3 Conclusion / Next Steps

I see work on SSLANG types taking a number of directions. With ADTs fully integrated into the compiler pipeline, we can start writing standard libraries, and even implement built-in types like booleans, characters, and strings with ADTS (provided they're paired with some front end syntax). Will SSLANG support some kind of arrays? What about packed data types, mentioned in my previous report? With the introduction of small built in types like characters, will we implement packed ssm values in codegen.hs to save heap space?

I hope to continue developing my SSLANG hackathon projects. With wheel3.ssl complete, I can see an ASCII animated flappy bird hopping on the SSLANG horizon.

References

1. Stephen A. Edwards and John Hui. The Sparse Synchronous Model. In Forum on Specification and Design Languages (FDL), Kiel, Germany, September 2020.
2. Leijen, Daan, et al. Prettyprinter, 2000, https://hackage.haskell.org/package/prettyprinter-1.7.1/docs/Prettyprinter.htmlg:2.
3. Mainland, Geoffrey. Language.C.Quote, 2010, https://hackage.haskell.org/package/language-c-quote-0.13/docs/Language-C-Quote.html.

Appendix

A colors.ssl

```
type Color
  White
  Black
  RGB Int Int Int

println ( cout : & Int ) ( n : Int ) -> () =
  after 10, cout <- n
  wait cout
  after 10, cout <- 10 // '\n'
  wait cout

printColor ( cout : & Int ) (c : Color) -> () =
  match c
    White = println cout 87           // 'W'
    Black = println cout 75           // 'K'
    (RGB 255 255 0) = println cout 89 // 'Y'
```

```
    (RGB 161 242 0) = println cout 71 // 'G'
    _ = println cout 63               // '?'

main ( cout : & Int ) -> () =
  let licorice = Black
  let lemon = RGB 255 255 0
  let lime = RGB 161 242 0
  printColor cout lime
  printColor cout lemon
  printColor cout licorice
```

Output:

```
G
Y
K
```

B mixingColors.ssl

```
type Color
  White
  Black
  RGB Int Int Int

println ( cout : & Int ) ( n : Int ) -> () =
  after 10, cout <- n
  wait cout
  after 10, cout <- 10 // '\n'
  wait cout

printColor ( cout : & Int ) (c : Color) -> () =
  match c
    (RGB 161 242 0) = println cout 71 // 'G'
    (RGB 255 147 b ) =
      match b
        0 = println cout 79           // 'O'
        142 = println cout 80         // 'P'
        255 = println cout 82         // 'R'
        _ = println cout 63           // '?'
    White = println cout 87           // 'W'
    Black = println cout 75           // 'K'
    _ = println cout 63 // '?'

main ( cout : & Int ) -> () =
  let print = printColor cout
  let lime = RGB 161 242 0
  let base = RGB 255 147
```

```
  print (base 0)          // orange
  print (base 255)        // rose
  print (base 142)        // peach
  print lime
```

Output:

```
O
R
P
G
```

C numbers4.ssl

```
// incomplete big number library
// - integer to big num conversion
// - big num printing complete
// - big num equality check
// - fixes numbers3.ssl by removes Nil from Number's ADT definition
// - TODO: successor function!

type Bool
    True
    False

type Digit
    Zero
    One
    Two
    Three
    Four
    Five
    Six
    Seven
    Eight
    Nine

type Number
    Nums Number Number
    Dig Digit

btos b : Bool -> Int =
    match b
          False = 70
          True = 84

dtoc (n : Digit) -> Int =
```

```
    match n
          Zero = 48
          One = 49
          Two = 50
          Three = 51
          Four = 52
          Five = 53
          Six = 54
          Seven = 55
          Eight = 56
          Nine = 57



eqZero n : Digit -> Bool =
    match n
          Zero = True
          _ = False

eqOne n : Digit -> Bool =
    match n
          One = True
          _ = False

eqTwo n : Digit -> Bool =
    match n
          Two = True
          _ = False

eqThree n : Digit -> Bool =
    match n
          Three = True
          _ = False

eqFour n : Digit -> Bool =
    match n
          Four = True
          _ = False

eqFive n : Digit -> Bool =
    match n
          Five = True
          _ = False

eqSix n : Digit -> Bool =
    match n
          Six = True
          _ = False
```

```
eqSeven n : Digit -> Bool =
    match n
          Seven = True
          _ = False

eqEight n : Digit -> Bool =
    match n
          Eight = True
          _ = False

eqNine n : Digit -> Bool =
    match n
          Nine = True
          _ = False

eqDigit (n : Digit) (n2 : Digit) -> Bool =
    match n
          Zero = eqZero n2
          One = eqOne n2
          Two = eqTwo n2
          Three = eqThree n2
          Four = eqFour n2
          Five = eqFive n2
          Six = eqSix n2
          Seven = eqSeven n2
          Eight = eqEight n2
          Nine = eqNine n2

itod (i :Int) -> Digit =
    match i
          0 = Zero
          1 = One
          2 = Two
          3 = Three
          4 = Four
          5 = Five
          6 = Six
          7 = Seven
          8 = Eight
          _ = Nine

itoNum (i : Int) -> Number =
    match (i > 10)
        0 = Dig (itod i)
        _ = let r = i % 10
            let q = i / 10
            Nums (itoNum q) (Dig (itod r))
```

```
printNum (cout : & Int) (n : Number) -> () =
    match n
      Dig dig = print cout (dtoc dig)
      Nums d tl = printNum cout d
                  printNum cout tl

eqNumber (n : Number) (n2 : Number) -> Bool =
    match n
      Dig d = match n2
                Dig d2 = eqDigit d d2
                _ = False
      Nums d tl = match n2
                    Nums d2 tl2 = match (eqNumber d d2)
                                    True = match (eqNumber tl tl2)
                                             True = True
                                             _ = False
                                    _ = False
                    _ = False

print (cout : & Int) (c : Int) -> () =
    after 10 , (cout : & Int) <- c
    wait (cout : & Int)

main (cint : & Int) ( cout : & Int ) -> () =
      let x = (Nums (Dig One) (Nums (Dig Nine) (Dig Nine)))
      let y = (Nums (Dig Five) (Nums (Dig Five) (Dig Zero)))
      print cout (btos (eqNumber  x x))
      print cout (btos (eqNumber x y))
      print cout 10
      printNum cout (Dig Nine)
      print cout 10
      printNum cout (itoNum 199)
      print cout 10
      printNum cout (itoNum 99)
      print cout 10
      printNum cout (itoNum 9)
      print cout 10
      printNum cout (itoNum 2199)
      print cout 10
      printNum cout y
      print cout 10
```

Output:

```
TF
9
```

```
199
99
9
2199
550
```