COMS 6901 E

Algebraic Data Types in SSLANG

Emily Sillars

12/22/21

Abstract

SSLANG (Sparse Synchronous Language) is a concrete implementation and extension of the Sparse Synchronous Model proposed by Edwards and Hui in the paper of the same name. A functional language taking inspiration from languages like Haskell and OCaml, one important feature in this extension of the SMM is the addition of algebraic data types. Since SSLANG is designed for use on low level hardware like micro controllers and embedded systems, it compiles to C code for maximum portability and fine-tuned control of hardware timers. Given an algebraic data type defined in SSLANG, what C code will be generated by the compiler? How will pattern matching and field access be implemented in C? How will these ADTs interface with SSLANG's memory management system? The following sections in this report aim to answer these questions.

1 Motivating Example: MyBool

An example ADT MyBool is defined using a haskell-esque syntax below. Given this ADT definition, what C code will be generated by the compiler?

```
data MyBool = MyTrue int | MyFalse int int
let a = MyFalse 42 18
```

At the simplest level, a sum type like MyBool can be thought of as having a one to one correspondence with a struct containing a tag field and a union of structs, something like

```
enum myBoolTag{MyTrue = 0, MyFalse};
struct MyTrue{
 int myTrue_0;
};

struct MyFalse{
 int myFalse_0;
 int myFalse_1;
};

struct MyBool{
 char tag;  // signifies whether object
            // is a MyTrue or a MyFalse
 union {
```

```
        struct MyTrue myTrue;
        struct MyFalse myFalse;
 }payload;
};
```

A logical next question might be, "where will this object be stored in memory?" The SSM runtime uses a simulated stack memory, meaning its activation records are stored in a queue allocated on the heap. From the perspective of a SSLANG programmer then, what does it mean to write the line `let a = MyFalse 42 18` ?

Does this mean a MyBool struct gets allocated within the program's activation record, on the simulated "Stack"? Or does this mean MyBool a gets allocated on the heap, with only a pointer to a stored within the program's activation record? The cost of dereferencing a pointer is significant. Perhaps a MyTrue instance should be stored on the stack. But then what about MyFalse? Is a data constructor with two integer fields small enough to be kept on the stack? How big does an object need to be before the cost of dereferencing a pointer outweighs the cost of storing the object in the activation record? Another downside of storing an object on the heap is the need to dereference it before viewing its tag field – this could lead to situations where the programmer frequently spends resources dereferencing a pointer to view a tag, only to discard the object because its not the right instance.

Compounding this issue are recursive data types, for example

```
data Tree a = TwoChildren Tree Tree | OneChild Bool Tree | Leaf
```

Here, to handle the cyclic association, the fields of type Tree within the tree data constructor must be of pointer types when represented as a C struct.

```
struct Twochildren{
 Tree* twochildren_0;         // left kid; must by pointer!
 Tree* twochildren_1;         // right kid; must by pointer!
};

struct Onechild{
    Bool onechild_0;
    Tree* onechild_1;         // only kid; must by pointer!
};
```

And then about the more general case, when an object contains another object as a field. Should that field always be a pointer? Consider this example MyCircle ADT:

```
data MyRadius = int
data MyPoint = Point int int
data MyCircle = Circle Radius MyPoint
```

Should the equivalent Circle struct contain a pointers to MyRadius and MyPoint structs, or can these structs be stored directly inside the Circle, since MyRadius, for example, is only an integer in size?

```
struct Circle{                    //option 1
    struct MyRadius* circle_0;
    struct MyPoint* circle_1;
};
```

```
struct Circle{                    //option 2
    struct MyRadius circle_0;
    struct MyPoint circle_1;
};
```

The root of this stack vs. heap issue comes down to the question of of what it means for an ADT to be "small' vs. "big". We decided to employ a memory model similar to the one used by Xavier Leroy in his ZINC compiler for ML [2] to answer this question.

In Leroy's model, any object larger than a single word is considered large, and any object less than or equal to word in size is considered small. Objects smaller than a word are rounded up to the size of a word. As a consequence, the smallest chunk of memory that will ever be referenced in the runtime system memory is a word in size. This means for 32 bit architectures (where a word is four bytes), the two least significant bits of any pointer will always be zero. Leroy takes advantage of these unused bits to distinguish pointers from word sized objects: if a word's least significant bit is zero, it's a pointer to a "large" ADT on the heap; if a word's least significant bit is one, it's a "small" ADT allocated directly on the runtime stack.

One disadvantage of this memory model is a loss in integer precision (31 bit integers instead of 32 bit integers). One advantage is a reduction in the number of pointer dereferences needed. For small objects, the bits for the tag and fields can be "packed" into a single word, resulting in constant time access to the tag through bitwise operations. Large objects still require a pointer dereference for access, but an integer or nullary data constructor will never be stored on the heap (excluding explicit references, of course). This is good. Finally, the attractiveness of such a simplistic memory model cemented our decision.

2 Interfacing with the Memory Manager

SSLANG employs a reference counting memory manager. This means the memory manager keeps track of the number of references pointing to each heap allocated object. When a reference to an object disappears, for example when a pointer to an object goes out of scope, the memory manager "drops" the object, decrementing its reference count. As soon as an object's reference count falls to zero, the memory manager must free up that object, as well as recursively drop any objects the freed object held references to.

Akin to Leroy's ZINC system once more, we decided to place a memory management header at the start of each ADT. (Small ADTs do not need a header since they're allocated on the stack, essentially enclosed inside a memory-managed activation record struct) . This header would be used by the memory manager to store reference count information about the object needed to do its job. We also needed a way for the memory manager to know which fields inside an object would need to be recursively dropped upon freeing that object.

At first, we considered rearranging the fields inside the ADTs, so fields that were pointers would all be grouped together at the top. This is a strategy outlined by Appel and Shao in the design of their ML compiler [3]. The idea here is the memory manager can start with the first field following the meta data header, and read linearly downward. At each field, it checks if it's a pointer, or integer. If it finds a pointer, the memory manager recursively drops the object pointed to. If it finds an integer, the MM can stop reading because the rest of the fields are non-pointers. While this strategy simplifies the memory manager's job, it increase the complexity of the compiler, which would have to rearrange the fields at the code generation level, and ensure the correct the mapping from pre-reordered fields to post-reordered ones. Instead, we decided to include the number of

fields inside the object in the meta data header. This way, the MM can read linearly downward starting with the first field after the header, and will know to stop after it has checked a particular number of fields. Finally, to keep the structure of large ADTs uniform, we decided to include the tag field inside the meta data header.

```
struct ssm_mm {
  uint8_t val_count; // number of fields in the ADT's payload.
  uint8_t tag;       // instance of the ADT (which data constructor)
  uint8_t ref_count; // number of references to this object
};
```

3 Adding 31 Bit Integer Support

After deciding on our memory model, we needed to change the representation of integers in SSLANG from 32 bit values to 31 bit values with the least significant bit set to 1. This change affected integer literals, all the arithmetic and nearly all the bitwise operations in the language. For example, multiplication needed to change from `a * b` to `marshal(unmarshal(a) * unmarshal(b))` , where unmarshal converts a 31 bit integer to a 32 bit integer:

```
unmarshal ( A ) = (A >> 1)
```

and marshal converts a 32 bit integer to a 31 bit integer:

```
marshal ( A ) = (A << 1) | 0x1
```

Certain operations, namely subtraction, addition, and bitwise negation can be translated into fewer assembly instructions than the result of applying the generic marshal and unmarshal functions. After adding the generic marshal and unmarshal functions to SSLANG's code generator, I added optimizations translations for these aforementioned operations. My work on this can be found in a merged PR here: https://github.com/ssm-lang/sslang/pull/41

Whether these optimizations would have a concrete effect on the performance of a SSLANG program as a whole is unclear. I performed some small tests comparing the generated assembly for optimized translations vs. general marshal/unmarshal, and my results were promising, but more testing is needed. Results of my tests can be found in section A of the appendix.

4 ADT Struct Definitions

Now that we have determined the memory model for ADTs and their interface with the memory manager, we can adapt our C struct translation of MyBool into a much more general form. Consider the following true statements and their corresponding C definitions.

"Every ADT in SSLANG is a word, and every word is either a pointer or an integer"

```
typedef union {
    struct ssm_mm *heap_ptr;
    ssm_word_t packed_val;
} ssm_value_t;
```

"'Big' ADTs are pointers to objects allocated on the heap. A 'big' ADT object contains a meta data header followed by some number of fields"

```
struct ssm_object {
    struct ssm_mm mm;
    ssm_value_t payload[1];
};
```

By making the payload an array of 0 values, we can use the same struct definition to represent any big ADT; we rely on the size in the metadata field and unsafe casting to access fields. Using these new generic definitions for any ADT we translate

```
data MyBool = MyTrue int | MyFalse int int
let a = MyFalse 42 18
```

into

```
enum MyBool{MyTrue = 0, MyFalse};
void main() {
    struct ssm_object* tmp = ssm_new(/* max size */ 2, /* tag */ MyFalse);
    tmp->payload[0] = (ssm_value_t) { .packed_val = ((42) << 1 | 1) };
    tmp->payload[1] = (ssm_value_t) { .packed_val = ((18) << 1 | 1) };
    ssm_value_t a = (ssm_value_t) { .heap_ptr = &(tmp)->mm };
}
```

`ssm_new` invokes the memory manager, which allocates space for the meta data header and ADT's fields and then correctly initializes the header.

Now that we know the C code we want to generate, how do we go about generating it? Given an ADT defintion consisting of a Type Constructor and Data constructor(s), C struct generation can be broken down into a few steps:

1. Determine whether each instance of an object is "small" or "big"; round up the size of "small" ADTs to a single word.
2. Generate a C enum for the ADT's tags, with "small" data constructors getting the smaller tag values (list the names of the "small" data constructors in the enum first before listing the names of the "big" data constructors.)
3. Save information about the ADT in a lookup table for later use in codegen.hs

where

- the size of a Data Constructor is the sum of its fields
- each field in an ADT is a word in size
- "small" is <= one word, "big" is > one word
- the lookup table of information is defined as

```
data TypeDefInfo = TypeDefInfo
{ dconType  :: M.Map DConId TConId
, typeSize  :: M.Map TConId Int
, isPointer :: M.Map DConId Bool
, tag       :: M.Map TConId (C.Exp -> C.Exp)
, intInit   :: M.Map DConId C.Exp
, ptrFields :: M.Map DConId (C.Exp -> Int -> C.Exp)
}
```

The final three fields in the look up table store snippets of generated C code unique to each ADT for extracting its tag and initializing its fields. Since an integer is the smallest built-in type SSLANG currently supports, the lookup table assumes small ADTs only contain nullary data constructors.

An edge case exists where a Data Constructor deemed "small" can sometimes end up larger than a word after incorporating its tag bits. To resolve this issue, as extra check between steps 1 and 2 is performed to calculate tag bits and ensure all the instances deemed "small" will indeed fit in a word. An illustration of this edge case is in section B of the Appendix.

5 ADT Initialization, Field Access and Pattern Matching

After defining and filling a lookup table with meaningful information in TypeDef.hs, I augmented the the underlying program state monad GenFnState in Codegen.hs to contain an instance of this table. One look up table is generated per call to genTypeDef, so genProgram combines the tables generated from all the ADT definitions/calls to genTypeDef into a single table stored in GenFnState. Later, when ADT initialization occurs in genExpr, generating the correct C code becomes a simple series of table look ups.

Pattern matching ADTs is equivalent to writing a C switch statement that switches on the tag field of an object. There are three variations of the extract function:

1. When an ADT only contains "small" instances, the extract tag function looks like

```
extractTag(A) = ((A._packed_int >> 0x1) & tagBits)
```

2. When an ADT only contains "big" instances, the extract tag function looks like

```
intTagVal(A) = (A.heap_ptr->mm.tag)
```

3. When an ADT contains both "small" and "big" instances, the extract tag function looks like

```
tagVal(A) = ((A & 0x1) ? intTagVal : ptrTagVal)
```

Categorizing ADTs in this way allows us to omit the "isInt" (A 0x1) check when unnecessary. My work on ADT intitialization, field access and pattern matching can be found in my changes to Codegen.hs and TypeDef.hs in an open PR here: https://github.com/ssm-lang/sslang/pull/50

6 Conclusion/Next Steps

At the time of writing this report, the IR representation of ADTs exists in the main branch, but not the syntax or parsing at the front end of the compiler. This means all of my changes to the code generation files (with the exception of 31 bit integer support) have not been tested. An urgent next step is to test my code on the MyBool example to make sure the correct C code is generated. This can be achieved by manually writing the MyBool example in IR and then running it through the rest of the compiler, or completing the front end portion the compiler for ADTs.

Adopting the "everything is a word" memory model required a substantial reorganization and rewriting of the SSLANG runtime system, implemented in Hui's PR "Support word-size values and memory management". In addition to testing, the adt-init-c-struct-gen branch will need to pull these PR changes and resolve any merge conflicts before approval.

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. Leroy, X. The Zinc Experiment: An Economical Implementation of the ML Language. INRIA, 1990.
3. Appel, Andrew W, and Zhong Shao. "A Type-Based Compiler for Standard ML." ACM SIGPLAN Notices, vol. 30, no. 6, June 1995, pp. 116–129.

Appendix

A 31 Bit Integer Marshal/Unmarshal Optimizations

I wrote two and three way addition functions, using both regular marshal/unmarshal operations, and an optimized marshal/unmarshal for addition.

```
int32_t add(int32_t a, int32_t b) {
    a = a >> 1;
    b = b >> 1;
    int32_t c = a + b;
    c = c << 1;
    c = c | 0x01;
    return c;
}
```

```
int32_t addOpt(int32_t a, int32_t b) {
    b = b & 0xFFFFFFFE;
    int32_t c = a + b;
    return c;
}
```

Figure 1: Two Way Addition in C

```
int32_t add3(int32_t a, int32_t b, int32_t c) {
    a = a >> 1;
    b = b >> 1;
    int32_t d = a + b;
    d = d << 1;
    d = d | 0x01;
    d = d >> 1;
    c = c >> 1;
    int32_t e = d + c;
    e = e << 1;
    e = e | 0x01;
    return e;
}
```

```
int32_t add3Opt(int32_t a, int32_t b, int32_t c) {
    b = b & 0xFFFFFFFE;
    int32_t d = a + b;
    c = c & 0xFFFFFFFE;
    int32_t e = d + c;
    return e;
}
```

Figure 2: Three Way Addition in C

I compared these versions by compiling the corresponding C code to assembly on godbolt.org and counting the number of instructions. I ran these comparisons with settings for a 32 bit and 64 bit ARM processor as well as a 64 bit x86 processor, with both CLANG and gcc compilers. I found that compiling with gcc on the optimized marshal/unmarshal consistently saved two assembly instructions.

32 BITS				
ARM64 gcc 11.2	**add -O2**	**addOpt -O2**	**add3 -O2**	**add3Opt -O2**
	add:	addOpt:	add3:	add3Opt:
	asrs r1, r1, #1	bic r1, r1, #1	asrs r1, r1, #1	bic r1, r1, #1
	add r1, r1, r0, asr #1	add r0, r0, r1	add r1, r1, r0, asr #1	bic r2, r2, #1
	lsls r0, r1, #1	bx lr	sbfx r1, r1, #0, #31	add r0, r0, r1
	orr r0, r0, #1		add r1, r1, r2, asr #1	add r0, r0, r2
	bx lr		lsls r0, r1, #1	bx lr
			orr r0, r0, #1	
			bx lr	
armv7-a clang 11.0.1	**add -O2**	**addOpt -O2**	**add3 -O2**	**add3Opt -O2**
	add:	addOpt:	add3:	add3Opt:
	bic r0, r0, #1	bic r1, r1, #1	bic r0, r0, #1	bic r1, r1, #1
	add r0, r0, r1	add r0, r1, r0	add r0, r0, r1	add r0, r1, r0
	orr r0, r0, #1	bx lr	bic r0, r0, #1	bic r1, r2, #1
	bx lr		add r0, r0, r2	add r0, r0, r1
			orr r0, r0, #1	bx lr
			bx lr	

Figure 3: 32 Bit Results

64 BITS				
ARM64 gcc 11.2	**add -O2**	**addOpt -O2**	**add3 -O2**	**add3Opt -O2**
	add:	addOpt:	add3:	add3Opt:
	asr w1, w1, 1	and w1, w1, -2	asr w1, w1, 1	and w1, w1, -2
	add w0, w1, w0, asr 1	add w0, w1, w0	add w0, w1, w0, asr 1	add w1, w1, w0
	lsl w0, w0, 1	ret	sbfx x0, x0, 0, 31	and w0, w2, -2
	orr w0, w0, 1		add w2, w0, w2, asr 1	add w0, w1, w0
	ret		lsl w0, w2, 1	ret
			orr w0, w0, 1	
			ret	
armv8-a clang 11.0.1	**add -O2**	**addOpt -O2**	**add3 -O2**	**add3Opt -O2**
	add: // @add	addOpt: // @addOpt	add3: // @add3	add3Opt: // @add3Opt
	and w8, w0, #0xfffffffe	and w8, w1, #0xfffffffe	and w8, w0, #0xfffffffe	and w8, w1, #0xfffffffe
	add w8, w8, w1	add w0, w8, w0	add w8, w8, w1	add w8, w8, w0
	orr w0, w8, #0x1	ret	and w8, w8, #0xfffffffe	and w9, w2, #0xfffffffe
	ret		add w8, w8, w2	add w0, w8, w9
			orr w0, w8, #0x1	ret
			ret	
x86-64 gcc 11.2	**add -O2**	**addOpt -O2**	**add3 -O2**	**add3Opt -O2**
	add:	addOpt:	add3:	add3Opt:
	sar edi	and esi, -2	sar edi	and esi, -2
	sar esi	lea eax, [rsi+rdi]	sar esi	and edx, -2
	add edi, esi	ret	add edi, esi	add esi, edi
	lea eax, [rdi+1+rdi]		sar edx	lea eax, [rsi+rdx]
	ret		lea eax, [rdi+rdx]	ret
			lea eax, [rax+1+rax]	
			ret	
x86-64 clang 13.0.0	**add -O2**	**addOpt -O2**	**add3 -O2**	**add3Opt -O2**
	add: # @add	addOpt: # @addOpt	add3: # @add3	add3Opt: # @add3Opt
	and edi, -2	and esi, -2	and edi, -2	and esi, -2
	lea eax, [rdi + rsi]	lea eax, [rsi + rdi]	lea eax, [rdi + rsi]	lea eax, [rsi + rdi]
	or eax, 1	ret	and eax, -2	and edx, -2
	ret		add eax, edx	add eax, edx
			or eax, 1	ret
			ret	

Figure 4: 64 Bit Results

B Edge Case: Many Nullary Data Constructors

```
data Color = RGB Char Char Char
      | CMYK Char Char Char Char
      | Red | Orange | Yellow | Green | Blue | Indigo | Violet
      | Black | White | Gray | Gold | Silver | Bronze | Rainbow
```

```
    let paint = RGB 52 158 235
```

All ADTs are represented with a single word.
1. Questions:
- Which must be a pointer to an object on the heap?
- Which can be a 31 bit integer instead?

2. Find candidate Data Constructors that contain few enough fields to fit in a single word

Data Constructor	sizeOf (bits)	decision
RGB	24	CANDIDATE INTEGER
CMYK	32	Pointer to Heap
Red	0	CANDIDATE INTEGER
Orange	0	CANDIDATE INTEGER
Yellow	0	CANDIDATE INTEGER
Green	0	CANDIDATE INTEGER
Blue	0	CANDIDATE INTEGER
Indigo	0	CANDIDATE INTEGER
Violet	0	CANDIDATE INTEGER
Black	0	CANDIDATE INTEGER
White	0	CANDIDATE INTEGER
Gray	0	CANDIDATE INTEGER
Gold	0	CANDIDATE INTEGER
Silver	0	CANDIDATE INTEGER
Bronze	0	CANDIDATE INTEGER
Rainbow	0	CANDIDATE INTEGER

3. Determine which candidates still fit into a single word when taking into account tag bits

There are 15 candidate data constructors, which means 8 bits will be needed for the tag.

Data Constructor	sizeOf + tag	decision
RGB	24 + 8 = 32	Pointer to Heap
Red	0 + 8 = 8	INTEGER
Orange	0 + 8 = 8	INTEGER
Yellow	0 + 8 = 8	INTEGER
Green	0 + 8 = 8	INTEGER
Blue	0 + 8 = 8	INTEGER
Indigo	0 + 8 = 8	INTEGER
Violet	0 + 8 = 8	INTEGER
Black	0 + 8 = 8	INTEGER
White	0 + 8 = 8	INTEGER
Gray	0 + 8 = 8	INTEGER
Gold	0 + 8 = 8	INTEGER
Silver	0 + 8 = 8	INTEGER
Bronze	0 + 8 = 8	INTEGER
Rainbow	0 + 8 = 8	INTEGER

4. Answers:
- RGB and CMYK must be pointers to objects on the heap
- all the rest are 31 bit integers

5. Generate Tag Enum with 31 bit integers getting the lowest tag values

```
enum colorTag { Red, Orange, Yellow, Green, Blue, Indigo, Violet, Black,
                White, Gray, Gold, Silver, Bronze, Rainbow, RGB, CMYK };
```

6. Generate struct initialization with payload containing max number of fields
The max number of fields a Color Data Constructor can have is 4 (CMYK).

```
void main() {
  struct ssm_object* tmp = ssm_new(/* max size */ 4, /* tag */ RGB);
  tmp->payload[0] = (ssm_value_t) { .packed_val = ((52) << 1 | 1) };
  tmp->payload[1] = (ssm_value_t) { .packed_val = ((158) << 1 | 1) };
```

```
  tmp->payload[1] = (ssm_value_t) { .packed_val = ((235) << 1 | 1) };
  ssm_value_t paint = (ssm_value_t) { .heap_ptr = &(tmp)->mm };
}
```