CSE 302: Compilers | Project

Structured and Aggregate Types

Starts: 2023-11-20

1 SYNOPSIS

In this project you will update the compiler you built in earlier labs to BX extended with support for aggregate data (arrays and structs) and the heap. You will then write a compiler for this to x64. The following will be expected of you.

- Extending the parser to handle BX pointers and arrays
- Type-checking BX pointers and arrays
- Extending TAC with `load`, and `store` instructions
- Emitting x64 for these extensions
- (Stretch) Add support for BX `struct`s.
- (Stretch) Add support for type-abbreviations and recursive types.

You are not required to perform any optimizations or use CFGs or SSA form in this project.

Warning

Stretch goals are worth extra credit, but should be attempted only after completing all other goals.

2 THE BX LANGUAGE

The BX language in this project is a strict superset of that of lab 4. It adds no new control structures, but does modify the type system and adds support for reading and writing from memory and structured data.

2.1 *Pointers and Variable-Length Arrays*

The BX type system so far consisted of exactly two types, `int` and `bool`. We now extend that with two more type forms: pointers and arrays. Given a type τ, the type τ`*` is the type of pointers to a chunks of memory that contain values of type τ.

$\langle$type$\rangle$ ::= ··· | $\langle$type$\rangle$ `'*'`

Allocation To allocate a new block of heap memory holding a finite array of values of type τ, BX adds a new `alloc` expression which take the type of the contents, τ, and the number of elements in the array as parameters.

$\langle$expr$\rangle$ ::= ··· | `'alloc'` $\langle$type$\rangle$ `'['` $\langle$expr$\rangle$ `']'`

The expression `alloc` τ`[`n`]` allocates n (which must be > 0 *at runtime*) contiguous blocks of size suitable to store values of type τ; this expression then itself has type τ`*`. The count argument n is mandatory: if you want to allocate a single slot, use `1` for n. The memory that is returned by `alloc` is 0'd out. Thus, the initial value stored in the allocated memory is whatever is represented with a sequence of 0-bytes.

This allocated memory persists, in effect, until the end of the lifetime of the program, although it is allowed for the BX runtime to detect when an allocated block of memory becomes inaccessible by starting from any live variables in the program and then subsequently to *free* that memory. *You don't have to implement memory reclamation for this project.*

Given a pointer-valued expression `e` of type τ`*`, the expression `*e` *dereferences* the pointer to return the value of type τ that is stored at the address `e`. Likewise, `e[`n`]` gets the element that is at slot n in the array allocated there. You may think of `*e` as a synonym for `e[0]`.

Assignment The expression forms `*e` and `e[`n`]` behave like variables and can be assigned to like any other variable. In fact there are a collection of such expressions that we call ⟨assignable⟩.[1]

```
⟨assignable⟩ ::= IDENT | '*' ⟨expr⟩ | ⟨expr⟩ '[' ⟨expr⟩ ']'
⟨expr⟩ ::= ··· | ⟨assignable⟩
```

Note that all ⟨assignable⟩s are ⟨expr⟩essions. Assignment statements now allow setting any assignable to a value of a compatible type.

```
⟨assign⟩ ::= ⟨assignable⟩ '=' ⟨expr⟩ ';'
```

Referencing Any assignable of type τ can be turned into a pointer (of type τ`*`) by means of the unary referencing operator, `&`.

```
⟨expr⟩ ::= ··· | '&' ⟨assignable⟩
```

This pointer can be used like any other pointer and can even be passed to other procedures and functions. However, note that pointers that point to the current stack frame will be automatically invalidated with the current call `return`s, so if these pointers are stored in data structures that persist past the lifetime of the current call then it is undefined what happens when those pointers are subsequently used. BX is not memory-safe: dereferencing invalid pointers can blow your stack and crash your program.

Null Pointers All pointers are allowed to be `null`, which is a special value that denotes an invalid pointer. It is a new kind of expression that can have any pointer type, i.e., the value `null` can be used to assign to any assignable of type τ`*`. It is a dynamic error to dereference `null`, which will immediately abort the program. From a representational standpoint, the bytes used to represent `null` are all 0-bytes.

```
⟨expr⟩ ::= ··· | 'null'
```

Note that `null` can be used both as an initial value and as arguments of pointer type.

[1] In C/C++ these are known as *lvalues*.

```
def f(p : int*) { }

def main() {
   var x = null : int*;     // can initialize int* with null
   f(null);                 // can use null in place of int* args
}
```

Pointer Operations The only allowed operators on pointers are == and `!=`. We have the following table for the expression `p == q` where both `p` and `q` are of the pointer type τ*:

`p` is `null`?	`q` is `null`?	`p == q`
yes	yes	`true`
no	yes	`false`
yes	no	`false`
no	no	`true` iff `p` and `q` point to the same thing

2.2 *Fixed-Size Arrays*

For a literal non-negative number $n \in \mathbb{N}$, the type τ`[`n`]` is the type of arrays of fixed size n containing elements of type τ. The benefit of fixed-size arrays is that their size is known at compile time, and hence when they are themselves used in aggregate structures it is not necessary to use extra pointers.

⟨type⟩ ::= · · · | ⟨type⟩ "[" ⟨number⟩ "]"

To access an element of an array, you would use square brackets for subscripting, just like for ordinary pointers. To allocate such an array on the heap, you would use an `alloc` expression as usual. Subscripting for fixed-size arrays is the same as for variable-sized arrays. Note that `alloc` expressions always produce pointers, never arrays:

```
// allocate a single int[10].
var x = alloc int[10][1] : int[10]*;

// allocate 20 blocks of int[10]s.
var y = alloc int[10][20] : int[10]*;
```

Initializing arrays Variables of type τ`[`n`]` are initialized with the single number `0`, which is interpreted to mean 0-ing out the contents of the array.

```
// allocate an array of 10 ints, all initialized to 0
var x = 0 : int[10];
```

Assignment In the assignment expression `l = e;` where `e` has type τ`[`n`]`, the effect is to copy each of the n elements of `e` into `l`.

2.3 *Structured Data (Stretch Goal)*

BX also allows for structured data with named fields. Such types are introduced with the `struct` keyword and define a list of unique mappings from field names to the types of the fields.

⟨type⟩ ::= ⋯ | 'struct' '{' ⟨struct-field⟩ (',' ⟨struct-field⟩)* '}'
⟨struct-field⟩ ::= IDENT ':' ⟨type⟩

The size of a `struct` type is the sum of the sizes of the types of the various fields of the `struct`. Each field also has a fixed *offset* from the start of the header, determined by the sizes of all the field types that come before the given field in the `struct`.

To illustrate, consider the following `struct`:

```
struct { f1 : int, f2 : int*, f3 : int[10], f4 : struct { g1 : bool } }
```

Its fields have the following sizes and offsets:

field	size (bytes)	offset (bytes)
`f1`	8	0
`f2`	8	8
`f3`	80	16
`f4`	8	96

The `struct` itself is 104 bytes in size.

Indexing and Assignment To access a given field of a `struct` we use `.`-notation like in C. Given an expression e of type

$$\texttt{struct \{ } \texttt{f}_1 : \tau_1, \texttt{f}_2 : \tau_2, \ldots, \texttt{f}_k : \tau_k \texttt{ \}}$$

the expression $\texttt{e.f}_i$ is an expression of type τ_i. This notation is also assignable. As a standard simplification of the syntax for subscripting structs, we use the notation e`->f` to stand for `(*`e`).f`.

⟨assignable⟩ ::= ⋯ | ⟨expr⟩ '.' ⟨variable⟩ | ⟨expr⟩ '->' ⟨variable⟩

Like with arrays, assigning a variable of struct type copies all the fields of the struct.

2.4 *Type Abbreviations and Recursive Types (Stretch Goal)*

Note that the `struct` types of BX are anonymous. This means that the only way to create recursive types is to introduce *type names* to the language. We do this by means of type *abbreviations*, which are a new kind of top-level declaration that introduces a new synonym for a given type.

⟨decl⟩ ::= ⋯ | ⟨type-abbrev⟩

⟨type-abbrev⟩ ::= 'type' IDENT '=' ⟨type⟩ ';'

Like other top-level declarations, type abbreviations are also mutually recursive with all the other type abbreviations; indeed, this is the only way we can define a recursive type such as a linked list, as the example below shows.

```
type list = struct {
  data : int,
  next : list*,
};
```

To compute the size of a type in the presence of type abbreviations, we sometimes need to look up the definition of the abbreviation. In the occurrence of `list*` we did not need to look up `list` since pointer types always have size 8 bytes. However, if we were to write the following recursive type, then we would have a cyclicity in the size calculation.

```
type bad_list = struct {
  data : int,
  next : bad_list,
};
```

Here, to compute the size of `bad_list` we will eventually need to know the size of `bad_list` itself. All types in BX are required to have sizes that can be determined without such cycles. The type `bad_list` above will therefore be considered to be ill-formed.

2.5 *Further* BX *Language Notes*

SENDING AND RECEIVING AGGREGATE TYPES Aggregate types (arrays and structs) may be used as desired in the body of a callable. However, it is *not allowed* to define a callable that has a parameter of aggregate type, nor is it allowed to return an aggregate type from a function. Note that you may send and receive *pointers* to aggregate types.

YE OLDE FIBONACCI EXAMPLE A CSE302 project won't be complete without an updated Fibonacci example. One is provided in figure 1 that makes use of some of the new features of BX. Note that it is far from comprehensive as tests go.

3 COMPILING BX

3.1 *Stack vs. Heap Allocation*

A temporary or local variable of aggregate type (array or struct) can be allocated on the stack just like a variable of non-aggregate type. These variables do not have initializing expressions,[2] but are instead 0'd out. To reinforce this, we will use the syntax `= 0` for initializing expressions for aggregate types, even though the literal numeber `0` is usually of integral type. Examples:

```
// an array of 10 integers, all 0
var x = 0 : int[10];

// a record with field f as 0, g as false (represented as 0)
var y = 0 : struct { f : int, g : bool };
```

For heap-allocation, on the other hand, the new memory is guaranteed to be 0'd out.

[2]Another planned feature of BX removed for time constraints.

```
type fibtree = struct {
  val  : int,
  prev : fibtree*[2],
};

def make_fibtree(n : int) : fibtree* {
  var ret = alloc fibtree[1] : fibtree*;
  if (n == 0) {
    ret->val = 0;    // actually redundant
  } else if (n == 1) {
    ret->val = 1;
  } else {
    ret->prev[0] = make_fibtree(n - 1);
    ret->prev[1] = make_fibtree(n - 2);
    ret->val = ret->prev[0]->val + ret->prev[1]->val;
  }
  return ret;
}

def main() {
  var ft : make_fibtree(20) : fibtree*;
  print ft->val;
}
```

Figure 1: Fibonacci Example in BX

```
// a pointer to a heap-allocated array of 10 integers, all 0
var x = alloc int[10] : int[10] *;

// a pointer to a heap-allocated struct with 0 fields.
var y = alloc struct { f : int, g : bool }
  : struct { f : int, g : bool } *;
```

When allocating temporaries of aggregate types on the stack, you will need to subtract a variable number of bytes from RSP. Thus, you need to change your compiler to track stack *offsets* rather than stack slot numbers for your pseudos and local variables. Moreover, since we require all such variables to be initialized with 0, you may want to make calls to `memset()` (or a similar function you write in your BX runtime) to zero out the stack. For example, see figure 2.

3.2 *Referencing and Assignment*

The key feature of an assignable is that it has a *memory location*. Variables are easy: they are just located at their corresponding data segment (for global variables) or stack slots (for local variables). For array subscripting (i.e., the `[]` notation), there is the additional cost of computing the offset into the array. Likewise, for structs it is necessary to compute the offset of the field that is accessed.

To facilitate memory accesses, we add two new instructions to TAC: `load` and `store` that make use of memory references in 4-tuple notation (explained below).

```
    .globl main
    .text
f:
    pushq %rbp
    movq %rsp, %rbp
    subq $96, %rsp

    # now zero out the newly allocated stack frame
    pushq %rdi              # save old rdi (which may be arg1 of f)
    pushq %rsi              # save old rsi (which may be arg2 of f)
    pushq %rdx              # save old rdx (which may be arg3 of f)
    leaq 24(%rsp), %rdi     # top of stack ignoring above 3 saves
    movq 0, %rsi            # the thing to fill with, i.e., 0
    movq $96, %rdx          # the number of bytes to set
    callq memset            # the C standard library's memset()
    popq %rdx               # restore arg3 of f
    popq %rsi               # restore arg2 of f
    popq %rdi               # restore arg1 of f
    # done zeroing out the stack

    # ... rest of the function
```

Figure 2: Zeroing out the allocated stack frame

TAC instruction	Description
`t = load (tb,ti,ns,no);`	Load contents of memory location determined by the 4-tuple `(tb,ti,ns,no)` into `t`. If `ti` has value 0, then both `ti` and `ns` can be omitted.
`store t, (tb,ti,ns,no);`	Store `t` into the memory location determined by the 4-tuple `(tb,ti,ns,no)` into `t`. If `ti` has value 0, then both `ti` and `ns` can be omitted.

The notation `(tb,ti,ns,no)` stands for the location:

$$\texttt{tb} + (\texttt{ti} * \texttt{ns}) + \texttt{no}$$

and requires `tb` and `ti` to be temporaries and `ns` and `no` to be `NUM64`s.

This format for memory locations is chosen to be inline with the full syntax for memory addresses in x64: `offset(%rbase,%rindex,scale)` where `offset` is a literal 32-bit integer (or a label for PC-relative addressing), `scale` $\in \{ 1, 2, 4, 8 \}$, and `%rbase` and `%rindex` are registers. All but `%rbase` are optional. The *effective address* of such an address is the value of the expression:

$$\texttt{\%rbase} + (\texttt{\%rindex} \times \texttt{scale}) + \texttt{offset}$$

Thus, for example, if RAX holds the address of an `int*` e, and RCX holds the number n, then `0(%rax,%rcx,8)` is *the address of* e`[`n`]`.

For computing with addresses, an important instruction is `leaq` which computes an effective address and stores *the address* into the destination register. For instance:

```
leaq 0(%rax,%rcx,8), %r10     # R10 = RAX + RCX * 8 + 0
```

would set `R10` to the above computed value. Then, if you want to actually dereference the contents at that address, you might use `movq` like so: `movq (%r10), %r11`.

4 DELIVERABLES

At the end of this project you should produce a program called `bxc.py` that would convert a BX source file specified in the command line, say `prog.bx`, into x64 assembly, `prog.s`. You may choose to emit the intermediate `prog.tac` if you wish (can be controlled with command line options).