15-411 Writeup 4

George Ralph (gdr) **Tuesday 9th November 2021**

1 Evaluation Order

1.

$$\begin{gathered} H; S; \eta \vdash \mathit{decl}(p, \mathit{int}*, s) \blacktriangleright K \\ \implies H; S; \eta[p \to \epsilon] \vdash \mathit{assign}(p, \text{NULL}) \blacktriangleright K \\ \implies H; S; \eta[p \to \epsilon] \vdash \text{NULL} \triangleright \mathit{assign}(p, \_), K \\ \implies H; S; \eta[p \to \text{NULL}] \vdash \text{Nop} \blacktriangleright K \\ \implies H; S; \eta[p \to \text{NULL}] \vdash \mathit{assign}(*p, 1/0) \blacktriangleright K \\ \implies H; S; \eta[p \to \text{NULL}] \vdash p \blacktriangleright \mathit{assign}(*\_, 1/0), K \\ \implies H; S; \eta[p \to \text{NULL}] \vdash 1/0 \blacktriangleright \mathit{assign}(*NULL, \_), K \\ \implies \mathit{exception}(\mathit{arith}) \end{gathered}$$

2.

$$\begin{gathered} H; S; \eta \vdash \mathit{decl}(p, \mathit{int}*, s) \blacktriangleright K \\ \implies H; S; \eta[p \to \epsilon] \vdash \mathit{assign}(p, \text{NULL}) \blacktriangleright K \\ \implies H; S; \eta[p \to \epsilon] \vdash \text{NULL} \triangleright \mathit{assign}(p, \_), K \\ \implies H; S; \eta[p \to \text{NULL}] \vdash \text{Nop} \blacktriangleright K \\ \implies H; S; \eta[p \to \text{NULL}] \vdash \mathit{asnop}(*p, +, 1/0) \blacktriangleright K \\ \implies H; S; \eta[p \to \text{NULL}] \vdash \&(*p) \blacktriangleright \mathit{asnop}(\_, +, 1/0), K \\ \implies H; S; \eta[p \to \text{NULL}] \vdash p \blacktriangleright \mathit{asnop}(\_, +, 1/0), K \\ \implies H; S; \eta[p \to \text{NULL}] \vdash NULL \blacktriangleright \mathit{asnop}(\_, +, 1/0), K \\ \implies H; S; \eta[p \to \text{NULL}] \vdash 1/0 \blacktriangleright \mathit{asnop}(NULL, +, \_), K \\ \implies \mathit{exception}(\mathit{arith}) \end{gathered}$$

3.

$$\begin{gathered} H; S; \eta \vdash \mathit{decl}(x, \mathit{int}[], s) \blacktriangleright K \\ \implies H; S; \eta[x \to \epsilon] \vdash \mathit{assign}(x, \mathit{alloc\_array}(\mathit{int}, 0)) \blacktriangleright K \\ \implies H; S; \eta[x \to \epsilon] \vdash \mathit{alloc\_array}(\mathit{int}, 0) \triangleright \mathit{assign}(x, \_), K \\ \implies H; S; \eta[x \to \epsilon] \vdash 0 \triangleright \mathit{alloc\_array}(\mathit{int}, \_), \mathit{assign}(x, \_), K \\ \implies H; S; \eta[x \to \epsilon] \vdash a \triangleright \mathit{assign}(x, \_), K \end{gathered}$$

$$\Longrightarrow\ H;S;\eta[x\to a]\vdash \mathrm{Nop} \blacktriangleright K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash asnop(x[0],+,1/0) \blacktriangleright K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash \&x[0] \rhd asnop(\_,+,1/0),K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash x \rhd \&\_[0], asnop(\_,+,1/0),K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash a \rhd \&\_[0], asnop(\_,+,1/0),K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash 0 \rhd \&a[\_], asnop(\_,+,1/0),K$$

$$\Longrightarrow\ exception(mem)$$

4.

$$H;S;\eta\vdash decl(x,structs*,s) \blacktriangleright K$$

$$\Longrightarrow\ H;S;\eta[x\to \epsilon]\vdash assign(x,\mathrm{NULL}) \blacktriangleright K$$

$$\Longrightarrow\ H;S;\eta[x\to \epsilon]\vdash \mathrm{NULL} \rhd assign(x,\_),K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash \mathrm{Nop} \blacktriangleright K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash assign(*(\&x.n),1/0) \rhd K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash \&(*x).n \rhd assign(*\_,1/0),K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash \&(*x) \rhd \&\_.n, assign(*\_,1/0),K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash x \rhd \&\_.n, assign(*\_,1/0),K$$

$$\Longrightarrow\ H;S;\eta[x\to \mathrm{NULL}]\vdash 0 \rhd \&\_.n, assign(*\_,1/0),K$$

$$\Longrightarrow\ exception(mem)$$

2 Polymorphism

1. There are three transition rules here. In the first, we evaluate e until it is a value. In the case where e evaluates to null, we tag the value with 0, so that we can identify and untag it later to any type. Otherwise, we will mark the first 8 bytes as our type tag, and the next 8 bytes with the pointer.

$$H; S; \eta\ tag(\tau*, e) \triangleright K \to H; S; \eta\ e \triangleright tag(\tau*, \_), K$$

$$H; S; \eta\ v \triangleright tag(\tau*, \_), K \to H[a \to tprep(\tau), a+8 \to v, next \to a+16]; S; \eta\ a \triangleright K(v \neq NULL)$$

$$H; S; \eta\ v \triangleright tag(\tau*, \_), K \to H[a \to 0, a+8 \to v, next \to a+16]; S; \eta\ a \triangleright K(v = NULL)$$

2. Like above, there are three transition rules. In the first, we evaluate the expression into a pointer. If the pointer is null, we raise a memory exception. Otherwise, we dereference the pointer and read the type annotation from memory, as well as the stored pointer. In the cases where the pointer is null or the type annotation matches the one provided, we return the pointer, otherwise, we raise a type exception.

$$H; S; \eta\ untag(\tau*, e) \triangleright K \to H; S; \eta\ e \triangleright untag(\tau*, \_), K$$

$$H[v \to w, v \to p]; S; \eta\ v \triangleright untag(\tau*, \_), K \to exception(memory)(v = NULL)$$

$$H[v \to w, v \to p]; S; \eta\ v \triangleright untag(\tau*, \_), K \to exception(type)(w \neq tprep(\tau), p \neq NULL)$$

$$H[v \to w, v \to p]; S; \eta\ v \triangleright untag(\tau*, \_), K \to H[a \to w, a \to p]; S; \eta\ p \triangleright K(w = tprep(\tau))$$

3. For $tag(\tau, e)$, a will contain our output

```
$cogen(e, p)$
$a \leftarrow malloc(16)$
$a' \leftarrow a + 8$
$M[a] \leftarrow 0$
$M[a'] \leftarrow 0$
$if(p = NULL)\ goto\ exit$
$M[a] \leftarrow tprep(\tau)$
$M[a'] \leftarrow p$
$exit:$
```

For $untag(\tau, e)$, p will contain our output

```
$cogen(e, a)$
$if(a = NULL)\ raise\ memexn$
$t \leftarrow M[a]$
$if(t! = tprep(\tau))\ raise\_tag$
$a' \leftarrow a + 8$
$p \leftarrow M[a']$
```

4. In the safe case, e_1 and e_2 are pointers to a data structure which contains the actual pointer and its type tag. Thus, if we tag the same pointer p twice, the resulting tagged pointers will point to two unique allocated locations in memory, and therefore will return false under a naive pointer comparison.

5. In the unsafe mode, we can simply use naive comparison, however in the safe mode, we will want to dereference the actual pointers (upper 8 bytes) of e_1 and e_2 and compare those to ensure both point to the same memory location or NULL.

3 Function Pointers

1. We would need to allow for function calls to include a left hand side expression which could evaluate to a pointer (ie a dereference star would be part of the non-terminal), in addition to the traditional symbol and arguments list we already have for function calls. Additionally, we would need to be able to parse function types for type declarations, and include a token for the ampersand operator so that we can identify accesses to the addresses of functions.

2. Our type system would need to be able to handle function types (this is, functions with both argument and return types) and we would need our typechecker to determine that for any function call, the expression to determine the function pointer would need to return a pointer to a function with argument types that match all the arguments on the right hand side. We also would need to ensure that any function calls by symbol refer to a function of that symbol name, not a local variable with that symbol name, even if that symbol is a function pointer (function pointers must be dereferenced). Additionally, we would need to use our function environment to determine the type of function pointers generated by the ampersand operator.

3. In our internal representation, we would need to modify function calls to support both call by symbol, and calls by pointer. In the latter case, function calls would need to generate a temp to hold the callee's pointer after its expression has been evaluated. Here would be a good place to add a NULL check for any function pointers so that we can raise a memory exception before calling the function.

4. Being able to call functions by pointer would require us to reserve at least one register during function calls to retain the pointer to the called function. Ideally, this would be a callee saved register as we cannot use any registers reserved function arguments. Because CALL supports 64 bit registers as an argument, our instruction selection will likely remain unchanged. To match c0 conventions though, we may need to include a NULL check to raise memory exceptions.