1 Overview

The goal of this project is to provide a framework in which users can redirect the control flow GHC-compiled Haskell programs. For example, if we have the following program

```haskell
main :: IO ()
main = print (2 + 3)
```

we can inject it with a library containing the following code

```haskell
plus :: Integer -> Integer -> Integer
plus a b = unsafePerformIO (plus' a b)
  where plus' a b = do
          let c = a + b
          putStrLn (show a ++ " + " ++ show b ++ " = " ++ show c)
          return c
```

transforming the standard output

```
5
```

to the following output

```
2 + 3 = 5
5
```

While the current state of the project does not allow for actual IO to be done within replacement function due to unsolved technical issues, it does allow us to modify the function definition. e.g. we are able to use

```haskell
plus :: Integer -> Integer -> Integer
plus a b = (toInteger . length) ([1..fromInteger a] ++ [1..fromInteger b])
```

to change the behavior of (+) for negative values.

2 GHC Basics

The Glasgow Haskell Compiler (GHC) executes in several stages, best described by the following diagram

```
                                                                           +---------+
                                                         LLVM backend /-->| LLVM IR |--\
                                                                      |   +---------+  | LLVM
                                                                      |                v
 +------------+ Desugar  +------+ STGify  +-----+ CodeGen  +-----+    |   NCG    +----------+
 | Parse tree |--------->| Core |-------->| STG |--------->| C-- |----+--------->| Assembly |
 +------------+          +------+         +-----+          +-----+    |          +----------+
                                                                      |                ^
                                                                      |      +---+     | GCC
                                                            C backend \----->| C |-----/
                                                                             +---+
```

We are particularly interested in the `STG` and `C--` intermediate representations of the Haskell code, because they are closely related to GHC's execution model and thus our ability to disturb GHC compiled code with foreign code.

The `STG` machine has several registers, which are completely described by `struct StgRegTable` within the GHC sources. Some of these registers are mapped to hardware registers, and on the `x86_64` architecture, the mapping is as follows:

```
BaseReg   -> r13
Sp        -> rbp
Hp        -> r12
R1        -> rbx
R2        -> r14
R3        -> rsi
R4        -> rdi
R5        -> r8
R6        -> r9
SpLim     -> r15
MachSp    -> rsp
F|D[1..6] -> xmm[1..6]
```

where `BaseReg` is of type `struct StgRegTable *`. This information is accessible via `void getregs(stg_regset_t *)` when inside a C function hooked from Haskell.

The mapping between `STG` and `C--` is a relatively trivial yet crucial mapping. In `STG`, expressions (variables) can consist of only a single function application so something like

```haskell
num = 1 + 2 + 3 + 4
```

will get transformed into

```haskell
num = let tmp = 1 + 2
          tmp' = tmp + 3
      in tmp' + 4
```

and finally converted into the following `C--`

```
num_entry() //  [R1]
        { info_tbl: [(c1ad,
                      label: num_info
                      rep:HeapRep static { Thunk })]
          stack_info: arg_space: 8 updfr_space: Just 8
        }
    {offset
      c1ad: // global
          _rpV::P64 = R1;
          if ((Sp + 8) - 48 < SpLim) (likely: False) goto c1ae; else goto c1af;
      c1ae: // global
          R1 = _rpV::P64;
          call (stg_gc_enter_1)(R1) args: 8, res: 0, upd: 8;
      c1af: // global
          (_c1aa::I64) = call "ccall" arg hints:  [PtrHint,
                                                  PtrHint]  result hints:  [PtrHint] newCAF(BaseReg, _rpV::P64);
          if (_c1aa::I64 == 0) goto c1ac; else goto c1ab;
      c1ac: // global
          call (I64[_rpV::P64])() args: 8, res: 0, upd: 8;
      c1ab: // global
          I64[Sp - 16] = stg_bh_upd_frame_info;
          I64[Sp - 8] = _c1aa::I64;
          R2 = GHC.Num.$fNumInteger_closure;
          I64[Sp - 40] = stg_ap_pp_info;
          P64[Sp - 32] = sat_s19o_closure;
          P64[Sp - 24] = sat_s19p_closure+1;
          Sp = Sp - 40;
          call GHC.Num.+_info(R2) args: 48, res: 0, upd: 24;
    }
}
```

with a bunch of extra information omitted. What is important to note is that `C--` is written in continuation passing style (CPS), where results are not consumed by the caller but rather consumed by a continuation passed to the function. An example would be

```
add_cps :: (Num a) => a -> a -> (a -> b) -> b
add_cps a b f = f $ a + b

print_three :: IO ()
print_three = add_cps 1 2 print
```

This is further evidenced by the frequent occurrences of `jmp [rbp]` that occur in the disassembly of every GHC compiled program. This makes debugging GHC compiled programs especially hard, because it is almost impossible to examine the usual “stack frame” because all arguments are on the stack and it is difficult to determine which entries are continuations and which entries are arguments. Some work may be possible in C, because we can maybe check if the potential info table pointed to by each entry is a good info table, but this work has not been done yet (alternatively, check if it’s a good closure).

What is good news is that we are able to redirect Haskell control flow back into C by overwriting a conditional jump that occurs within the stack overflow check. The actual implementation is covered in `void hook(void *src, void *dst)`, but the overall goal is to rewrite

```
haskell_stub:
  lea rax, [rbp - $local_stack_space]
  cmp rax, r15
  jb .garbage_collection
  ; ...
.garbage_collection:
  ; ...
```

into

```
haskell_stub:
  lea rax, [rbp - $local_stack_space]
  cmp rax, r15
  jmp hook_stub
  ; ...
.garbage_collection:
  ; ...
```

where `hook_stub` will save the GHC runtime state, call into a C hook, restore the runtime state when the C hook returns, and continue executing the Haskell code. More good news is that as long the `STG` arity and type signature of two functions agree, we are able to retarget one stub into the other at no additional cost, because the runtime will expect everything to be laid out in the same way and everything else will sail smoothly. Unfortunately, this does mean that we cannot overwrite typeclass-defined functions with functions with a typeclass in the signature. In particular, the `(+) :: Num a => a -> a -> a` from `Num` has arity 1 while `plus' :: Num a => a -> a -> a` has arity 3. We can jump through some hoops and define a new typeclass and replace the typeclass “virtual function table” (*vtable*) in a hook stub, but that seems overly complicated. Instead, we overwrite entries in the typeclass *vtable* to point to our own Haskell code and that is sufficient.

Recall the assignment `R2 = GHC.Num.$fNumInteger_closure` from the `C--` code. `GHC.Num.+_info` is a stub that will look up the required `(+)` function and then return into `stg_ap_pp_fast` (apply function to 2 pointers) to apply the correct function to the two arguments. The `GHC.Num.$fNumInteger_closure` is actually an array that corresponds to the definition of the `Num` typeclass, containing closures to their respective functions. In particular, we have

```
-- | Basic numeric class.
class  Num a  where
    {-# MINIMAL (+), (*), abs, signum, fromInteger, (negate | (-)) #-}
```

```haskell
    (+), (-), (*)       :: a -> a -> a
    negate              :: a -> a
    abs                 :: a -> a
    signum              :: a -> a
    fromInteger         :: Integer -> a

    {-# INLINE (-) #-}
    {-# INLINE negate #-}
    x - y               = x + negate y
    negate x            = 0 - x

-- | @since 2.01
instance  Num Integer  where
    (+) = plusInteger
    (-) = minusInteger
    (*) = timesInteger
    negate         = negateInteger
    fromInteger x  =  x

    abs = absInteger
    signum = signumInteger
```

and

```
(lldb) disas -s `*(uint64_t *)(0x1000e46f0)` -c 1
base_GHCziNum_CZCNum_con_info:
    0x10003c9a0 <+0>: inc    rbx
(lldb) disas -s `*(uint64_t *)(*(uint64_t *)(0x1000e46f0+0x08) & ~0x07)` -c 1
integerzmgmp_GHCziIntegerziType_plusInteger_info:
    0x10007c748 <+0>: lea    rax, [rbp - 0x18]
(lldb) disas -s `*(uint64_t *)(*(uint64_t *)(0x1000e46f0+0x10) & ~0x07)` -c 1
integerzmgmp_GHCziIntegerziType_minusInteger_info:
    0x10007d250 <+0>: lea    rax, [rbp - 0x18]
(lldb) disas -s `*(uint64_t *)(*(uint64_t *)(0x1000e46f0+0x18) & ~0x07)` -c 1
integerzmgmp_GHCziIntegerziType_timesInteger_info:
    0x10007bda0 <+0>: lea    rax, [rbp - 0x18]
(lldb) disas -s `*(uint64_t *)(*(uint64_t *)(0x1000e46f0+0x20) & ~0x07)` -c 1
integerzmgmp_GHCziIntegerziType_negateInteger_info:
    0x10007a348 <+0>: lea    rax, [rbp - 0x10]
(lldb) disas -s `*(uint64_t *)(*(uint64_t *)(0x1000e46f0+0x28) & ~0x07)` -c 1
integerzmgmp_GHCziIntegerziType_absInteger_info:
    0x10007a580 <+0>: lea    rax, [rbp - 0x8]
(lldb) disas -s `*(uint64_t *)(*(uint64_t *)(0x1000e46f0+0x30) & ~0x07)` -c 1
integerzmgmp_GHCziIntegerziType_signumInteger_info:
    0x10007b020 <+0>: lea    rax, [rbp - 0x8]
(lldb) disas -s `*(uint64_t *)(*(uint64_t *)(0x1000e46f0+0x38) & ~0x07)` -c 1
base_GHCziNum_zdfNumIntegerzuzdcfromInteger_info:
    0x10003c978 <+0>: mov    rbx, r14
(lldb) disas -s `*(uint64_t *)(*(uint64_t *)(0x1000e46f0+0x40) & ~0x07)` -c 1
error: error: ; SNIPPED
```

Inside the demo project, this is accomplished with the assignment

```c
((void **)numinteger)[1] = dlsym(RTLD_SELF, "UnsafeZZp_unsafezuzzpzqzqzq_closure");
```

3 The main issue

One of the biggest issues surrounding this entire project when we introduced foreign Haskell code is the introduction of a separate GHC runtime. Since GHC really likes statically linking everything, this means that when we inject an application with foreign Haskell we will end up with two separate copies of the runtime, all of which are completely unrelated to each other. Because we wish to do interesting things with `unsafePerformIO :: IO a -> a`, which calls into the runtime, some work must be done to make sure that the two runtimes behave well with each other.

We first initialize our local runtime by calling

```
hs_init(&argc, &argv);
```

with some bogus data, but this is not enough. `unsafePerformIO` will return into the runtime, and more specifically, it will return into the functions

1. `maybePerformBlockedException`
2. `stg_unmaskAsyncExceptionszh_ret_info`
3. `tryWakeupThread`

While some of these are avoidable, we especially do not want to go into `tryWakeupThread` in our use case, it requires forcing our local runtime to refer back to the original binary's runtime, at the expense of rewriting some assembly instructions. In particular, we want to overwrite things of the form

```
cmp    rcx, qword ptr [rip + 0x40af82]
  ; (void *)0x0000000101c707d8: stg_END_TSO_QUEUE_closure
```

so that the `stg_END_TSO_QUEUE_closure` is the one in the original binary. Luckily, all instances of this pattern refer to the same pointer, so we can do this in one overwrite. The more complicated case occurs when we compare via `lea`.

```
lea    rdi, [rip + 0x45e0d1]      ; stg_END_TSO_QUEUE_closure
cmp    rbx, rdi
je     0x10181276c                ; <+140> at RaiseAsync.c:593
```

So far, only one replacement is required within `maybePerformBlockedException` to get non-IO replacements to run without crashing, and trying actual IO has given me a headache but will likely require more replacements elsewhere. Other times, when we try to do IO, we hit the `default` case in `evacuate(StgClosure **)`, which is because `HEAP_ALLOCED_GC` is not returning the correct value possibly because things are being allocated in the address spaces of two different runtimes. Unfortunately, because `USE_LARGE_ADDRESS_SPACE` is set by default in GHC compiled programs, this is a macro and will require significant effort to fix, either by messing with the allocator in the local runtime or by patching `evacuate`.

4 Results

We are able to patch Haskell binaries to return potentially different values and we are able to use `unsafePerformIO` as long as no actual IO is being done. In particular, within the demo application of a postfix stack calculator, we can replace `Integer` addition with

```
unsafe_zp''' :: Integer -> Integer -> Integer
unsafe_zp''' a b = unsafePerformIO $ do
  c <- return . toInteger . length $ [1..fromInteger a] ++ [1..fromInteger b]
  return $ c + 1
```

as a very wonky (and wrong) addition to produce the following output:

```
welcome to the postfix stack machine calculator
the PID for this process is 32920 for informational purposes
> 4 2 + 9000 +
9006
(injection happens)
> 4 2 + 9000 +
9008
>
```