Heap operations such as local allocation and object promotion sometimes must happen in C. Unfortunately, these operations can trigger local and global garbage collections respectively, and thus require special handling in the code generator. This note first introduces the problem with promotion, expands the problem to calls to C functions that allocate, and then proposes a solution.

We implement object promotion as a C function in the RTS, and will probably keep it that way. This strategy has the downside, however, that a promotion could trigger a global collection, and in general, it is probably too expensive to determine that a global collection is necessary until we've transferred into RTS code. For the example below, the size of the list we're promoting could exceed the free space in the global heap.

```hs
val ls = promote (List.tabulate (n, fn x => x))
```

Since promote can trigger a global collection, we need to set up the root set before transferring control to the RTS. The code we generate for promotion will be similar to the local GC transfer, and should be a straightforward change to the code generator. But the CFG representation for promotion (`<code>E_Promote</code>`) is a RHS form, making it difficult to generate this transfer code. Promotions should instead be transfers in the CFG, similar to the `<code>HeapCheck</code>` form.

Promotions only perform global allocations but arbitrary C functions can, at present, perform local allocations. We probably will want to disallow such local allocations because handling minor collections would be rather complex. If, for instance, we trigger a minor collection from C land, we would need to walk the C stack to build the root set. The fact that we use local copying collectors makes it even trickier because we would have to re-walk the C stack after minor collections. We can avoid this latter stack walk though by restricting C functions to "global" allocations. We can avoid the former stack walk by introducing a special transfer form for C calls that allocate, e.g.,

```
AlloCCall of
```

where `<code>f</code>` is the C function, `<code>args</code>` are its arguments, `<code>ret</code>` is the jump target after the C call finishes. By convention, we pass the return value of the ccall as the first argument of `<code>ret</code>`.

The pseudocode below generates such a C call.

```
allocate the root set, which is just retKArgs
store the root pointer and dedicated registers in the vproc data structure
retVal := ccall f (args)
```

```hs
restore the root set
```

```
throw retK (retVal, retKArgs ...)
```

Where `<code>f</code>` is the C function, `<code>args</code>` are its arguments, `<code>ret</code>` is the jump target after the C call finishes. By convention, we pass the return value of the ccall as the first argument of `<code>ret</code>`.