CMSC 22620 **Implementation** **Project 3**
Spring 2009 **of** **April 28, 2009**
Computer Languages

Garbage collection support

Due: May 10, 2009

1 Introduction

LangF is a heap-based language with automatic garbage collection (GC). In this project, you will extend your code generator to support garbage collection.

In this handout, we describe the way that heap objects are allocated and the interface between the code you generate and the garbage collector.

2 Allocation

String and array objects are allocated in runtime-system routines. Tuples, however, are allocated using straight-line code generated by the compiler. The `%rbx` register points to the next free heap word. To allocate a tuple, one writes the header word and then initializes the tuple's fields. Lastly, one puts the address of the object in a register (or memory location) and increments the allocation pointer. For example, the following code allocates a pair, where `%r13` and `%r14` hold the elements of the pair:

```
movq  $777,(%rbx)        // initialize header word
movq  %r13,8(%rbx)       // initialize first element
movq  %r14,16(%rbx)      // initialize second element
leaq  8(%rbx),%r15       // set %r15 to point to the object
leaq  24(%rbx),%rbx      // increment allocation pointer
```

The header value 777 corresponds to the bit pattern (11000001001), which specifies a 2-element tuple with both elements being potential pointers.

3 Garbage collection

While small LangF programs may be able to run to completion without requiring garbage collection (GC), many programs will eventually exhaust the space set aside for object allocation. When this event occurs, the garbage collector is called to reclaim dead objects. In this section, we describe the mechanism for detecting that GC is required, the protocol for invoking the GC, and the mechanism used to communicate liveness information to the GC.

3.1 Limit checks

The Cluster IR includes GC test operations. We keep the limit pointer (`%r12`) set to 8,192 bytes below the top of the heap. Thus, as long as a GC test requires fewer than 8,192 bytes, we can use a simple comparison between the allocation and limit pointers as in the following code fragment:

```
        cmpq  %r12,%rbx
        ja    L27
L28:
        ...

L27:    invoke GC
        jmp   L28
```

Note that we put the GC code below the test and make the conditional jump go to the GC code. This approach has two benefits:

1. The x86-64 branch prediction hardware predicts that forward jumps are likely to not be taken.[1] Since GC is the uncommon case, want to make sure that the jump to it is predicted to be not taken.

2. By locating the GC invocation code away from the main code, we reduce instruction-cache pollution.

3.2 GC roots

In order for the garbage collector to be able to correctly identify live objects, it must know the set of *roots*. These are registers and memory locations that possibly contain pointers into the heap; those objects that are reachable from the roots are deemed live and are not collected; all other objects are garbage and are reclaimed. The root set includes the registers that are live (and contain LangF values) when the GC is invoked and those stack locations that contain live values. It is the compiler's job to communicate this information to the collector as described below.

3.3 Communicating stack-location liveness

To communicate information about the liveness of stack locations, we use a static table of liveness records that is generated at compile time. This table contains a record for each non-tail call site in the program.[2] The beginning of this table is a 64-bit header word that holds the number of records in the table. This word should be labeled by the global label `LANGF_LiveTbl`. Each liveness record has the following format:

```
struct {
    Addr_t    pc;       // Return PC of call site
    uint32_t  stkSz;    // size of stack frame in 8-byte words
    uint32_t  live[n];  // live mask
};
```

[1] Note that backward jumps are predicted taken.

[2] Note that records for tail calls are *not* required.

The first element is the return address of the given call site; the second is the size of the stack frame;[3] and the third element is the lieveness mask with one bit per word in the frame. The liveness mask is padded so that the total size of the record is a multiple of eight bytes; *i.e.*, $n = 2 \times \left\lfloor \frac{\texttt{stkSz}+31}{64} \right\rfloor + 1$. Bits are numbered from the least significant to most significant, so the ith word will be represented by bit $i \mod 32$ of word $\left\lfloor \frac{i}{32} \right\rfloor$.

The words in a stack frame are numbered from the lowest address; thus the saved frame pointer will be word $\texttt{stkSz} - 2$ and the return address will be word $\texttt{stkSz} - 1$. For example, assume that we have a call site in a function whose frame size is six words (four words plus the linkage).

```
        call    foo
L12:
        ...
```

Further assume that words 0, 1, and 3 are live at the return. Then the liveness table will look something like the following (assuming that there are 23 call sites in the entire program):

```
        .p2align 3
LANGF_LiveTbl:
        .quad   23

        ...

        .p2align 3
        .quad   L12
        .long   6
        .long   11
```

The liveness mask is `001011`, which is decimal 11. Note that by forcing 64-bit alignment of the beginning of the descriptor, we get the assembler to pad the previous descriptor out to a multiple of 8 bytes.

3.4 Invoking the garbage collector

The garbage collector is invoked by calling the assembly routine `CallGC`, which is responsible for saving and restoring the registers across the GC. When invoking the collector, one must communicate liveness information about the current function (*i.e.*, the top-most stack frame). For the stack frame, we include a liveness record for the call to `CallGC` as above. We also need to pass information about the liveness of registers, which is done by passing a bit mask in `%r12`.[4] Thus, the *invoke GC* code for the above limit-check example will be as follows:

```
L27:
        movq  $mask,%r12
        call  CallGC
L29:
        jmp   L28
```

Here *mask* is the live-register mask and `L29` is the label for the liveness-table entry for this call.

Note that the garbage collector is only interested in those roots that potentially hold heap pointers (*i.e.*, values with boxed or mixed representation). Raw and tagged integer values do not have to be tagged as live for the collector (note that `CallGC` will preserve all registers, so these values will

[3] The frame size will always be a multiple of two, since stack frames must be 16-byte aligned.

[4] On return from GC, `%r12` will have been restored to the heap limit pointer.

be unchanged by the GC).

4 Computing liveness

You will need to compute liveness information for each GC test point and each non-tail application site in the cluster representation. This computation is a fairly simple backward flow analysis; since fragments are acyclic, you do not even need to iterate to a fixed point. We have modified the cluster representation to include a property-list holder on each expression; this holder can be used to record the live-variable sets.

Revision history

May 7, 2009 Fixed equation for calculating the number of words in the liveness mask so that it matches the semantics of integer arithmetic.
Also updated the due date.

April 28, 2009 Original version.