Assignment 4
Memory Layout and Polymorphism

15-411: Compiler Design
Frank Pfenning
Flávio Cruz, Maxime Serrano, Rokhini Prabhu, Tae Gyun Kim

Due Thursday, October 23, 2014 (23:59pm)

Reminder: Assignments are individual assignments, not done in pairs. The work must be all your own. Please hand in your solution electronically in PDF format and refer to the late policy for written assignments on the course web page.

Problem 1: Memory Layout (25 points)

Consider the following C0 source code on the left and the assembly code produced by your compiler on the right. Note that it uses tail-call optimization to avoid a recursive call.

```
struct d64 {
  int lo32;
  int hi32;
};

struct bigint {
  bool sign;
  struct d64 d;
  struct bigint* next;
};

void inc(struct bigint* q)
//@requires q != NULL && !q->sign;
{
  (*q).d.lo32 += 1;
  if ((*q).d.lo32 == 0) {
    (*q).d.hi32 += 1;
    if ((*q).d.hi32 == 0) {
      if (q->next == NULL)
        /* init's fields appropriately */
        q->next = alloc(struct bigint);
      inc(q->next);
    }
  }
  return;
}
```

```
_c0_inc:
        pushq   %rbx
        movq    %rdi, %rbx
        jmp     L11
L7:
        movq    12(%rbx), %rbx
L11:
        movl    4(%rbx), %eax
        incl    %eax
        movl    %eax, 4(%rbx)
        testl   %eax, %eax
        jne     L9
        movl    8(%rbx), %eax
        incl    %eax
        movl    %eax, 8(%rbx)
        testl   %eax, %eax
        jne     L9
        cmpl    $0, 12(%rbx)
        jne     L7
        movl    $20, %esi
        movl    $1, %edi
        call    calloc
        movq    %rax, 12(%rbx)
        jmp     L7
L9:
        popq    %rbx
        ret
```

(a) Explain why the compiler assigned variable q to the register `%rbx`.

(b) `calloc` takes arguments of type `size_t`, which expands to an `unsigned long int` and is therefore 64 bits wide, according to the x86-64 ABI. Why is it correct to use `movl` instructions instead of `movq` to set the argument registers?

(c) The assembly code does not conform to the x86-64 ABI. Explain why not and provide a correction.

(d) The assembly code contains a further bug. Identify it and provide a correction. Do not be concerned about whether the source program might have a bug; we are only concerned with whether the assembly code correctly matches the source.

(e) We are compiling in production mode, ignoring contracts. The given assembly code relies on OS memory protection in order to signal an error in case the argument to `_c0_inc` is the null pointer 0. Insert an appropriate check *in one place* that avoids relying on OS memory protection. You may assume a jump target `raise_mem` that will raise the appropriate memory exception. Briefly explain the rationale for your choice.

Problem 2: Polymorphism (35 points)

The C0 language provides only a very weak form of polymorphism, essentially using `struct s*` in a library header, where `struct s` has not yet been defined. C provides a more expressive, but inherently unsafe mechanism by allowing pointers of type `void*`. A pointer of this type can reference data of any type. We then use implicit or explicit casts to convert to and from this type. Some discussion and examples can be found in the notes on Lecture 19 in the course on *Principles of Imperative Computation*. In this problem we explore a safe version of `void*` which implements dynamic checking of polymorphic types and has made its way into C1.

Tagging and Untagging Data

The key to making the type $\mathsf{void}*$ safe is to tag pointers of this type with their actual type. When we cast values of this type to actual types we can then compare tags to make sure the operation is type-safe. We have new tagging and untagging constructs

$$e \ ::= \ \ldots \mid \mathsf{tag}(\tau*, e) \mid \mathsf{untag}(\tau*, e)$$

with the following typing rules

$$\frac{\Gamma \vdash e : \tau*}{\Gamma \vdash \mathsf{tag}(\tau*, e) : \mathsf{void}*} \qquad \frac{\Gamma \vdash e : \mathsf{void}*}{\Gamma \vdash \mathsf{untag}(\tau*, e) : \tau*}$$

Tagging is always safe: we can forget that e references a value of type τ and just weaken its type to $\mathsf{void}*$. Untagging will signal a runtime error if the tag of e is different from $\tau*$. For example, if $p : \mathsf{int}*$ then the expression

$$\mathsf{untag}(\mathsf{bool}*, \mathsf{tag}(\mathsf{int}*, p))$$

will type-check, but should yield a runtime error while untagging since $\mathsf{bool}* \neq \mathsf{int}*$.

A Safe Implementation

In the safe implementation, a value of type $\mathsf{void}*$ will always be either null (0), or a pointer to 16 bytes of memory on the heap. The first 8 bytes represent the actual type $\tau*$, the second 8 represent the actual value of type $\tau*$, which must be an address. We assume we can calculate $\mathsf{tprep}(\tau*) = w$, where w is a 8-byte tag value uniquely representing the type $\tau*$. The default value for type $\mathsf{void}*$ is null (0).

(a) Provide the evaluation rules for $\mathsf{tag}(\tau*, e)$. You should define new transition rules for the abstract machine with state $H \ ; \ S \ ; \ \eta \vdash e \rhd K$ as defined in lecture. Your rules do not need to check whether memory is exhausted. You should also describe the evaluation of $\mathsf{tag}(\tau*, e)$ informally, which will help us assign partial credit in case your rules are not entirely correct.

(b) Provide the evaluation rules for `untag`$(\tau*, e)$. This should fail if the tag of e does not match $\tau*$, in which case you should raise a `tag` exception. You should define new transition rules for the abstract machine as in part (a), and accompany them with an informal description.

(c) Describe code generation for the `tag` and `untag` expression forms in the style we used for arrays on page L14.7 of the lecture notes. You may use function calls

$$t^{64} \leftarrow \mathsf{malloc}(s^{64})$$

to obtain the address t of s bytes of uninitialized memory, and use the jump target `raise_tag` to signal a tag exception.

An Unsafe Implementation

The unsafe implementation should forego tag checking. As a result, we do not need to tag or untag at all, since we trust the programmer that tags would have been correct. In other words, `tag`$(\tau*, e)$ would be like `(void*)e` in C, and `untag`$(\tau*, e)$ like `(tau*)e`, relevant only at the type-checking phase.

(d) Explain why compiling e_1 `==` e_2 for pointers e_1 and e_2 to a naive pointer comparison is not always correct in *safe* mode.

(e) Explain how to compile e_1 `==` e_2 in both safe and unsafe modes so that program behavior is the same for both modes (assuming, of course, that the program is indeed safe and will not raise an exception). Code is not necessary if the implementation is clear enough from your description.