Assignment 4: Memory

15-411/611: Course Staff

Due Tuesday, November 9, 2021 (11:59pm)

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 pages.

Problem 1: Evaluation Order (20 points)

For each of the code snippets below, provide a trace using dynamic semantics rules from lecture to determine wha the correct outcome is. (See the lecture notes on mutable store and structs for the updated dynamic semantic rules, be careful about the += operation)

(a)

```
int* p = NULL;
*p = 1/0;
```

(b)

```
int* p = NULL;
*p += 1/0;
```

(c)

```
int [] x = alloc_array(int, 0);
x[0] += 1/0;
```

(d)

```
struct s {
  int n;
};
struct s* x = NULL;
x->n = 1/0;
```

Problem 2: Polymorphism (25 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. The programmer uses 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 runtime tag-checking of types—which, incidentally, is the approach taken in C0's successor C1.

Tagging and Untagging Data

The key to making coercions from the `void*` type-safe is to tag pointers of type `void*` with the contained data's type. When the runtime encounters a cast from type `void*` to another pointer type, the tag is checked to ensure that the cast is safe.

In the source language, we introduce new tagging and untagging constructs:

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

with the following typing rules

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

Tagging will never cause an error: regardless of the type of a pointer value, we can always weaken its type to `void*` and create a tag. Untagging a value (as in $\mathtt{untag}(\tau*, v)$) should raise a runtime error if v is the result of tagging a non-null pointer with a type differing from $\tau*$. For example, if $p : \mathtt{int}*$ is a non-null value, then the following is an expression that will typecheck but whose evaluation will raise a runtime error:

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

Untagging the result of tagging a null pointer should succeed regardless of the type the null pointer is tagged with. For example, the evaluation of this expression should succeed:

$$\mathtt{untag}(\mathtt{bool}*, \mathtt{tag}(\mathtt{int}*, \mathtt{NULL}))$$

A Safe Implementation

In the safe implementation, a value p of type $\mathtt{void}*$ will always be either null (0), or a pointer to 16 bytes of memory on the heap. The first 8 bytes on the heap are the tag for the type $\tau*$, and the second 8 contain a representation for p (which is an address).

Assume we have a function $\mathtt{tprep}(\tau)$, which takes as argument a type τ and returns an 8-byte tag w uniquely representing τ[1]. The default value for type $\mathtt{void}*$ is null (0).

(a) Provide the evaluation rules for $\mathtt{tag}(\tau*, e)$. You will define new transition rules for the abstract machine with state $H \,;\, S \,;\, \eta \vdash e \rhd K$ as defined in the lecture on mutable store. At least some of your transitions will involve allocation on the heap H.

You should also describe the evaluation of $\mathtt{tag}(\tau*, e)$ informally, which will help us assign partial credit in case your rules are not entirely correct.

[1]This is problematic in the sense that C0 allows for unboundedly many unique types to be defined, but let's pretend that there is a limit of 2^{64}.

(b) Provide the evaluation rules for `untag`$(\tau*, e)$. This should fail if e evaluates to a non-null value v whose tag does not match `tprep`$(\tau*)$, 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 in the lecture on mutable store. You may use function calls

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

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, there is no runtime computation performed for tagging or untagging. In other words, tags and untags are like casts in C, which are relevant only for type-checking.

The semantics of equality is as follows: for p_1, p_2 : `void*`, p_1==p_2 should evaluate to `true` if p_1 and p_2 are the result of tagging the same memory location. (This comparison should additionally evaluate to `true` if p_1 and p_2 are either `NULL` or the result of tagging `NULL`.) Otherwise, the comparison should evaluate to `false`.

(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. Recall that naive pointer comparisons are done by comparing addresses.

(e) Explain how to compile e_1 == e_2 in both safe and unsafe modes so that program has the same observable behavior for both modes (assuming 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.

Problem 3: Function Pointers (20 points)

Function pointers, or some variant of them, are a common language feature that is especially prevalent in C and C++. You might even like the slogan *"Function Pointers are Values"*. You probably remember passing a function pointer as an argument to `signal()` from 15-213, or using them to for client callbacks in advanced data structures in 15-122. Suppose we wanted to give C0 programmers the ability to declare variables as function pointers, assign functions to them, pass them as parameters to functions, call them, and even return them. Together with the polymorphism afforded by `void*`, this allows us to implement generic data structures in a (hypothetical, at this point) C1 language.

We will use a subset of the standard C syntax to declare and use function pointers. In order to do so, we will need to add the `&` (address-of) operator to the language. In C1 it will only be allowed to obtain the address of a function. In order to call a function pointer, it must first be dereferenced. Therefore, if f is a function pointer, $f()$ is illegal, but $(*f)()$ is allowed. For example, the following code would return 27 from `main`.

```
typedef
int i2i(int x);

typedef
int ii2i(int x, int y);

int succ(int x) {
  return x+1;
}
int plus(int x, int y) {
  return x+y;
}
int times(int x, int y) {
  return x*y;
}

int main() {
  i2i* s = &succ;
  ii2i* add = &plus;
  ii2i* mult = &times;
  int x = (*mult)(3,(*add)(succ(6),(*s)(1)));
  return x;
}
```

When answering the questions below, please keep in mind that there may be different design choices. Explain on your choices and make sure your approach in the answers to the questions is consistent.

(a) Describe, on a high level, what changes you would need to make to the grammar to handle function pointers. What nonterminals would need to be changed? What restriction do you impose compared to C while allowing the example above?

(b) Describe any changes to the static semantics, including the type checker, that would be needed to handle function pointers.

(d) Describe how you would modify the internal representation to allow function pointers. Describe which variables are used and defined by the modified IR instructions.

(e) Describe how adding function pointers would impact register allocation and instruction selection.

(f) **[Extra Credit]** Extend the design to allow contracts on function pointers which remains compatible with C and is consistent with your approach above and the general philosophy behind the design of C0.