# Assignment 4: Memory and Dynamic Semantics

15-411/611: Course Staff

Due Tuesday, March 18, 2025 (11:59PM)

**Reminder:** Assignments are individual assignments, not done in pairs. The work must be all your own. Hand in your solutions on Gradescope. Please read the late policy for written assignments on the course web page.

## Problem 1: Memory Semantics (10 points)

Consider the code snippet below:

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

And an elaborated abstract syntax tree of the program:

```
  declare(x, int[],seq (assign(x,alloc_array(int, 0)),
    asnop(x[0], plus, binop(1, div, 0)))
  )
```

Provide a trace using dynamic semantics rules from lecture to determine what the correct outcome is. If a step has side conditions (e.g. $n \geq 0$, etc.) explicitly justify them[1]. Your trace should have exactly 10 steps.

$$
\begin{array}{ll}
 & H; \cdot; \cdot \vdash \\
 & \texttt{declare}(\texttt{x}, \texttt{int[]}, \texttt{seq}(\texttt{assign}(x, \texttt{alloc\_array}(\texttt{int}, 0)), \texttt{asnop}(\texttt{x[0]}, \texttt{plus}, \texttt{binop}(1, \texttt{div}, 0)))) \blacktriangleright \cdot \\
\rightarrow & ?; ?; ? \vdash ? \\
\rightarrow & \ldots
\end{array}
$$

[1]See the lecture notes on mutable store and structs for the updated dynamic semantic rules, be careful about the += operation.

## Problem 2: Enums (20 points)

Many programming languages contain enumerations or sets of named constants. These enum constructs appear in languages such as C, C++, and Java, among others.

In C, enumeration types $u$ can be declared as

$$\texttt{enum}\ u;$$

or defined as

$$\texttt{enum}\ u\ \{v_1, \ldots, v_n\};$$

where $v_1, \ldots, v_n$ are distinct identifiers, and $u$ is an identifier. Enum values are introduced by named constants $v_i$, which are now valid expressions. Enum values can be used in switch statements, which take the form

$$\texttt{switch}(e)\{v_1 \mapsto s_1 \mid \ldots \mid v_n \mapsto s_n\}$$

Informally, a switch statement inspects the enum value that $e$ evaluates to and branches accordingly. In the above example, if $e$ steps to the constant $v_1$, then the statement $s_1$ will be executed. If $e$ steps to $v_2$, then $s_2$ will be executed. The pattern continues.

Below are a couple of rules that begin to describe the static semantics of enumerations.

$$\frac{?}{\Sigma; \Gamma \vdash \texttt{switch}(e)\{v_1 \mapsto s_1 \mid \ldots \mid v_n \mapsto s_n\} :?}\ \text{(S1)} \qquad \frac{?}{\Sigma; \Gamma \vdash v :?}\ \text{(S2)}$$

The rules use an enumeration signature $\Sigma$ that contains all defined enumerations. You can assume that every enumeration $u$ and every element $v$ appears at most once in the signature.

$$\Sigma ::= \cdot \mid \texttt{enum}\ u\ \{v_1, \ldots, v_n\};, \Sigma$$

(a) Complete the type rules for enumerations to maintain the type safety of C0. Hint: one thing that the premises for the rule S1 should check is that the named constants $v_1, \ldots, v_n$ are distinct and exhaustive.

(b) Extend the dynamic semantics for expressions and statements to describe the evaluation of named constants and the execution of switch statements. You should only need two rules.

## Problem 3: Polymorphism (20 points)

The C0 language has very few mechanisms for polymorphic function definitions. 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. 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:

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

with the following typing rules

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

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 $\texttt{untag}(\tau*, v)$) 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 $p : \texttt{int}*$ is a non-null value, then the following is an expression that will typecheck but whose evaluation will raise a runtime error:

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

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:

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

## A Safe Implementation

In our safe implementation, a value $p$ of type `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 `tagof`$(\tau)$, which takes as argument a type $\tau$ and returns an 8-byte tag $w$ uniquely representing $\tau$ [2]. The default value for type `void*` is null (0).

(a) Provide the formal dynamic semantics for $\texttt{tag}(\tau*, e)$. Your answer should consist of one or more transition rules. At least one of the rules should have the form

$$H; S; \eta \vdash \texttt{tag}(\tau*, e) \rhd K \quad \to \quad ?; ?; ? \vdash ?$$

Some of your transitions will involve allocation on the heap $H$.

(b) Provide the formal dynamic semantics for $\texttt{untag}(\tau*, e)$. As with part (a), your answer should consist of one or more transition rules. At least one of the rules should have the form

$$H; S; \eta \vdash \texttt{untag}(\tau*, e) \rhd K \quad \to \quad ?; ?; ? \vdash ?$$

[2] Formally, C0 allows for unboundedly many unique types to be defined, but let's pretend that there is a limit of $2^{64}$.