# Assignment 4: Semantics

15-411: Compiler Design

Due Sunday, November 3, 2019 (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: Generalized Ifs (15 points)

In this problem, assume we're using a subset of the restricted abstract syntax used in lecture, and the corresponding statics and dynamics. For your convenience, these are reproduced below.

### Language

$$
\begin{array}{llll}
\text{Operators} & \oplus & ::= & + \mid < \\
\text{Expressions} & e & ::= & n \mid x \mid e_1 \oplus e_2 \mid e_1 \texttt{\&\&} e_2 \\
\text{Statements} & s & ::= & \texttt{assign}(x, e) \mid \texttt{if}(e, s_1, s_2) \mid \texttt{while}(e, s) \\
 & & & \mid \texttt{return}(e) \mid \texttt{nop} \mid \texttt{seq}(s_1, s_2) \mid \texttt{decl}(x, \tau, s)
\end{array}
$$

### Statics

$$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 < e_2 : \texttt{bool}} \qquad \frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_2 : \texttt{bool}}{\Gamma \vdash e_1 \texttt{\&\&} e_2 : \texttt{bool}}$$

$$\frac{\Gamma(x) = \tau' \quad \Gamma \vdash e : \tau'}{\Gamma \vdash \texttt{assign}(x, e) : [\tau]} \qquad \frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \texttt{if}(e, s_1, s_2) : [\tau]}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash s : [\tau]}{\Gamma \vdash \texttt{while}(e, s) : [\tau]} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \texttt{return}(e) : [\tau]}$$

$$\frac{}{\Gamma \vdash \texttt{nop} : [\tau]} \qquad \frac{\Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \texttt{seq}(s_1, s_2) : [\tau]}$$

$$\frac{\Gamma, x{:}\tau' \vdash s : [\tau]}{\Gamma \vdash \texttt{decl}(x, \tau', s) : [\tau]}$$

## Dynamics

$$\begin{array}{lcl}
\eta \vdash e_1 \oplus e_2 \rhd K & \to & \eta \vdash e_1 \rhd (_ \oplus e_2, K) \\
\eta \vdash c_1 \rhd (_ \oplus e_2, K) & \to & \eta \vdash e_2 \rhd (c_1 \oplus _, K) \\
\eta \vdash c_2 \rhd (c_1 \oplus _, K) & \to & \eta \vdash c \rhd K \qquad (c = c_1 \oplus c_2) \\
 & & \\
\eta \vdash e_1 \texttt{\&\&} e_2 \rhd K & \to & \eta \vdash e_1 \rhd (_\texttt{\&\&}e_2, K) \\
\eta \vdash \texttt{false} \rhd (_\texttt{\&\&}e_2, K) & \to & \eta \vdash \texttt{false} \rhd K \\
\eta \vdash \texttt{true} \rhd (_\texttt{\&\&}e_2, K) & \to & \eta \vdash e_2 \rhd K \\
 & & \\
\eta \vdash x \rhd K & \to & \eta \vdash \eta(x) \rhd K \\
 & & \\
\eta \vdash \texttt{assign}(x, e) \blacktriangleright K & \to & \eta \vdash e \rhd (\texttt{assign}(x, _), K) \\
\eta \vdash c \rhd (\texttt{assign}(x, _), K) & \to & \eta[x \mapsto c] \vdash \texttt{nop} \blacktriangleright K \\
 & & \\
\eta \vdash \texttt{decl}(x, \tau, s) \blacktriangleright K & \to & \eta[x \mapsto \texttt{nothing}] \vdash s \blacktriangleright K \\
 & & \\
\eta \vdash \texttt{if}(e, s_1, s_2) \blacktriangleright K & \to & \eta \vdash e \rhd (\texttt{if}(_, s_1, s_2), K) \\
\eta \vdash \texttt{true} \rhd (\texttt{if}(_, s_1, s_2), K) & \to & \eta \vdash s_1 \blacktriangleright K \\
\eta \vdash \texttt{false} \rhd (\texttt{if}(_, s_1, s_2), K) & \to & \eta \vdash s_2 \blacktriangleright K \\
 & & \\
\eta \vdash \texttt{while}(e, s) \blacktriangleright K & \to & \eta \vdash \texttt{if}(e, \texttt{seq}(s, \texttt{while}(e, s)), \texttt{nop}) \blacktriangleright K \\
 & & \\
\eta \vdash \texttt{return}(e) \blacktriangleright K & \to & \eta \vdash e \rhd (\texttt{return}(_), K) \\
\eta \vdash v \rhd (\texttt{return}(_), K) & \to & \texttt{value}(v)
\end{array}$$

Thinking about C, Jan realizes how convenient it would be to have conditionals operate on any type by implicitly casting them to booleans. For example, we would expect the code fragment

```
if (7) { do_something_fun(); }
else { do_something_not_fun(); }
```

to call `do_something_fun()` in C, as 7 is non-zero. However, in C0 we only have a judgement for when the expression being compared upon is a boolean. To solve this problem,

Jan adds a new typing rule

$$\frac{\Gamma \vdash e : \texttt{int} \quad \Gamma \vdash s_1 : [\tau] \quad \Gamma \vdash s_2 : [\tau]}{\Gamma \vdash \texttt{if}(e, s_1, s_2) : [\tau]}$$

However, when he runs a small program using the semantics, the program gets stuck.

```
if (7) {
    return 1;
} else {
    return 0;
}
```

1. What could be wrong?
2. Provide a trace in the format from lecture exposing the problem.
3. Help Jan out and provide a fix for this issue that will allow `if` statements to function as he desires. Ensure that your fix does not break any other features of this language.

## 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

$$\mathsf{enum}\ u;$$

or defined as

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

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

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

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.

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

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.

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

(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.

## Problem 3: 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:

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

with the following typing rules

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

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

(a) Provide the evaluation rules for $\texttt{tag}(\tau*, e)$. You will define new transition rules for the abstract machine with state $H\ ;\ S\ ;\ \eta \vdash e \triangleright K$ 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 $\texttt{tag}(\tau*, e)$ informally, which will help us assign partial credit in case your rules are not entirely correct.

(b) Provide the evaluation rules for $\texttt{untag}(\tau*, e)$. This should fail if $e$ evaluates to a non-null value $v$ whose tag does not match $\texttt{tprep}(\tau*)$, 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.

[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 $2^{64}$.