Assignment 4: Semantics

15-411: Compiler Design
Jan Hoffmann, Vijay Ramamurthy, Prachi Laud, Nick Roberts, Shalom Yiblet

Due Thursday, November 1, 2018 (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

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

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 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 $v_1, \ldots, v_n$, and u are identifiers. Enums 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 u and branches based its value. In the above example, if e is the constant v_1, then the statement s_1 will be executed. If e is v_2, then s_2 will be executed, and so on. Expressions can now contain named constants v.

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

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

The rules use an enumeration signature Σ that contains all defined enumerations. You can assume that every enumeration u appear 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.

(b) Extend the dynamic semantics for expressions and statements to describe the evaluation of enumerations.

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. 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 `void*` safe is to tag pointers of type `void*` with their actual type. When we cast values of type `void*` to actual types we can then check the tag to make sure the operation is type-safe. We have new tagging and untagging constructs

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

with the following typing rules

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

Tagging will never cause an error: regardless of the type of pointer e, we can always weaken its type to `void*` and create a tag. Untagging will signal a runtime error if the tag of e is different from the type $\tau*$ that it is being cast to. For example, if p : `int*` then the expression

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

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

A Safe Implementation

In the safe implementation, a value of type `void*` will always be either null (0), or a pointer to 16 bytes of memory on the heap. The first 8 bytes represent its original type $\tau*$, the second 8 represent the actual value referenced by the pointer, which must be an address. We assume we can calculate

$$\texttt{tprep}(\tau*) = w$$

where w is a 8-byte tag value uniquely representing the type $\tau*$. The default value for type `void*` is null (0).

(a) Provide the evaluation rules for $\texttt{tag}(\tau*, e)$. You should define new transition rules for the abstract machine with state $H\ ;\ S\ ;\ \eta \vdash e \rhd K$ as defined in lecture 15.

Your rules do not need to check whether memory is exhausted. 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 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 in lecture 14. 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, we do not need to tag or untag at all, since we trust the programmer that tags would have been correct. In other words, $\texttt{tag}(\tau*, e)$ would be like casting `(void*)e` in C, and $\texttt{untag}(\tau*, e)$ like casting $(\tau*)$`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. 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 resulting 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.
*Note that $\texttt{tag}(\texttt{void*}, e)$ is statically disallowed in C0.