Assignment 4: Semantics

15-411: Compiler Design
Jan Hoffmann
Evan Bergeron, Xue An Chuang, Aaron Gutierrez, Shyam Raghavan

Due Thursday, November 3, 2016 (9:00am)

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

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

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

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

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

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

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

Dynamics

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

Thinking about C, Shyam 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, Shyam 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 compiles a small program using the feature and tries to run it, his program refuses to terminate.

```
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 Shyam 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.
   (Note: there are solutions to this that involve fixing either the statics or dynamics or both. Simple (but correct!) approaches are preferred).

Problem 2: C_0^λ Revisited (10 points)

After Aaron's language C_0^λ failed to gain traction among programmers, Evan stepped in to help. As you may recall, C_0^λ has the following syntax:

$$\begin{aligned} \text{Type } \tau &::= \text{int} \mid (\tau \times \tau \to \tau) \\ \text{Expression } e &::= n \mid x \mid (\lambda(x, y).e) \mid (e \oplus e) \mid (e(e, e)) \end{aligned}$$

You should be familiar with the expression $\lambda(x, y).e$, though perhaps not in the same format. In SML, this would be written as `fn (x, y) => e`.
Once he read these rules, Evan realized that Aaron forgot to do something – he never specified the semantics for his new language! Noticing that the typing judgements for this language are fairly simple, Evan quickly wrote the following rules:

$$\frac{}{\Gamma \vdash n : \text{int}} \qquad \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash (e_1 \oplus e_2) : \text{int}}$$

$$\frac{\Gamma \vdash e_1 : (\tau_1 \times \tau_2 \to \tau_3) \quad \Gamma \vdash e_2 : \tau_1 \quad \Gamma \vdash e_3 : \tau_2}{\Gamma \vdash e_1(e_2, e_3) : \tau_3} \qquad \frac{x : \tau_1, y : \tau_2 \vdash e : \tau_3}{\Gamma \vdash \lambda(x, y).e : (\tau_1 \times \tau_2 \to \tau_3)}$$

However, these rules are significantly more limiting than you might first think.

1. Give a short program that one might want to write but are unable to write in this language. `Ocaml` or `SML` would be good languages to provide examples in. *(Note: this is not related to the number of arguments functions take).*
2. What might be difficult if you were to implement these statics? *(Hint: think about modes).*

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 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

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

with the following typing rules

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

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

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

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

A Safe Implementation

In the safe implementation, a value 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 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 $\mathtt{tprep}(\tau*) = w$, where w is a 8-byte tag value uniquely representing the type $\tau*$. The default value for type $\mathtt{void}*$ 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.

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 `(void*)e` in C, and $\texttt{untag}(\tau*, e)$ 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.