Announcements

1. Assignment 6 distributed by Tuesday, 2/25; due 3/6 (10 days)

2. Midterm on 3/4; open books and notes
The Pond type system

Today we discuss the static type analysis for Pond:

- Motivations
- How it differs from C
- Formal definition

In assignment 6, you will implement this type system.
The Dangerous World of C

The following program compiles under gcc *without complaint*:

```c
int f(int *x)
{
    int *a;
    int i;
    short j;

    a = (int*)&f;
    *a = 0;
    f(a); /* now what happens? */

    i = 100000000;
    j = i;    /* undetected overflow; what’s the result? */

    for (i = 0; i < 10000; i++)
    {
        a++;
    }
    *a = 0; /* what did we overwrite? */
}
```
The Dangerous World of C

Facts:

- The “no frills” definition of C contributes to its speed
- The ill-defined behavior of C, permitted by its “type system”, is one of the most significant hazards in modern computing
  - example: the majority of security holes on Apache webservers are buffer-overflow attacks

Our goal: to see how types can make PLs safer (if not completely safe).
The Pond language: expressions

For our reference (from assignment 5):

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>$\in$ ID</td>
</tr>
<tr>
<td>$n$</td>
<td>$\in$ $\mathbb{Z}$</td>
</tr>
<tr>
<td>$c$</td>
<td>$\in$ ASCII</td>
</tr>
<tr>
<td>$e$</td>
<td>::= $x \mid n \mid c \mid e \ op \ e \mid e_1 \ op_a \ e \mid op_{ul} \ e \mid e \ op_{ur} \mid e \ [e] \mid e \ (e, \ldots, e) \mid (e)$</td>
</tr>
<tr>
<td>$e_1$</td>
<td>::= $x \mid * e \mid e \ [e]$</td>
</tr>
<tr>
<td>$op$</td>
<td>::= $+ \mid - \mid * \mid / \mid % \mid &lt; \mid &gt; \mid == \mid &lt;= \mid &gt;= \mid != \mid &amp;&amp; \mid</td>
</tr>
<tr>
<td>$op_a$</td>
<td>::= $= \mid += \mid -= \mid *= \mid /= \mid %=$</td>
</tr>
<tr>
<td>$op_{ul}$</td>
<td>::= $! \mid ++ \mid -- \mid - \mid * \mid &amp;$</td>
</tr>
<tr>
<td>$op_{ur}$</td>
<td>::= $++ \mid --$</td>
</tr>
</tbody>
</table>

NB: No type casts.
The Pond language: base types

For our reference; slightly modified from assignment 5 (same language, additional syntactic categories):

\[
\begin{align*}
\tau_i & ::= \text{int} | \text{short} | \text{long} | \text{char} & \quad \text{basic scalar types} \\
\tau_u & ::= \tau_i | \text{unsigned } \tau_i & \quad \text{unqualified scalar types} \\
\text{qual} & ::= \text{const} | \text{volatile} & \quad \text{type qualifiers} \\
\tau_m & ::= \tau_u | \text{volatile } \tau_u & \quad \text{mutable scalar types} \\
\tau_s & ::= \tau_u | \text{qual } \tau_u & \\
\tau_b & ::= \tau_i | \text{void} & \quad \text{scalar types} \\
\end{align*}
\]
The Pond language: declarations, statements, programs

For our reference (from assignment 5):

\[ b \ ::= \ \{ ds \ ss \} \]
\[ s \ ::= \ b \mid for \ (e; e; e) \ s \mid while \ (e) \ s \mid \]
\[ \quad if \ (e) \ s \ else \ s \mid switch \ (e) \ b \mid \]
\[ \quad case \ e : s \mid default : s \mid break; \mid \]
\[ \quad continue; \mid return; \mid return \ e; \mid e; \mid ; \]
\[ ss \ ::= \ \varepsilon \mid ss \ s \]
\[ var \ ::= \ x \mid * var \mid var[n] \mid (var) \]
\[ d \ ::= \ \tau_b \ var, \]
\[ ds \ ::= \ \varepsilon \mid ds \ d \]
\[ pd \ ::= \ \tau_b \ var \]
\[ ps \ ::= \ pd \mid ps, pd \]
\[ f \ ::= \ \tau_b \ x (ps) \ b \mid \tau_b \ x () \ b \mid \tau_b \ * x (ps) \ b \mid \tau_b \ * x () \ b \]
\[ pp \ ::= \ f \mid d \]
\[ p \ ::= \ \varepsilon \mid p \ pp \]
Derived type language

Although the Pond source language syntax comprises only base types explicitly, program objects imply a richer type language for our analysis:

\[
\tau ::= \tau_b \mid \tau \rightarrow \tau \mid \tau\text{array}[n] \mid \tau\text{ptr} \mid \tau\ast\tau \quad \text{derived types}
\]

- \(\tau \rightarrow \tau\): function types
- \(\tau\text{ptr}\): pointer types
- \(\tau\text{array}[n]\): array types
- \(\tau\ast\tau\): parameter types

Hereafter, we restrict parameter types to the domain types of functions
Type environments

To keep track of variable bindings imposed by declarations, we define type environments:

\[ \Gamma ::= \emptyset | \Gamma; x : \tau \]

**Definition 1** We define type environment lookup, denoted \( \Gamma(x) \), inductively as follows:

\[
\begin{align*}
(\Gamma; x : \tau)(x) &= \tau \\
(\Gamma; y : \tau)(x) &= \Gamma(x) \quad x \neq y
\end{align*}
\]
Obtaining type bindings from declarations

As seen in assignment 5, bindings of derived type forms are implicit in declarations of the form $\tau_b \text{var}$.

The construction of bindings from declarations can be formalized as follows:

- $\text{binding}(x, \tau_b) = x : \tau_b$
- $\text{binding}(\star \text{var}, \tau_b) = \text{let } x : \tau = \text{binding}(\text{var}, \tau_b) \text{ in } x : \tau \text{ptr}$
- $\text{binding}(\text{var}[n], \tau_b) = \text{let } x : \tau = \text{binding}(\text{var}, \tau_b) \text{ in } x : \tau \text{array}[n]$
Obtaining environments

With a method for obtaining a single binding, we can formalize a method of generating environments from sequences of declarations:

\[
\begin{align*}
\text{bindings}(\varepsilon) & = \emptyset \\
\text{bindings}(\tau_b \ var) & = \text{binding}(\text{var}, \tau_b) \\
\text{bindings}(ps, (\tau_b \ var)) & = \text{bindings}(ps); \text{binding}(\text{var}, \tau_b) \\
\text{bindings}(ds (\tau_b \ var;)) & = \text{bindings}(ds); \text{binding}(\text{var}, \tau_b)
\end{align*}
\]

Note that bindings is defined on both decl. sequences and parameter lists.
Subtype compatibility: scalars

For flexibility in our language, it is convenient to treat some datatypes as being compatible with others.

If \( \tau \leq \tau' \), then a value of type \( \tau \) can be \textit{used} as a value of \( \tau' \) \textit{in allowable contexts}.

\[
\text{char} \leq \text{short} \quad \text{short} \leq \text{int} \quad \text{int} \leq \text{long}
\]

NB: the relation only goes one way, to ensure well-definedness.
Subtype compatibility: scalars

\[
\begin{align*}
\tau & \leq \tau' \\
\text{unsigned } \tau & \leq \text{unsigned } \tau' \\
\text{unsigned } \tau & \leq \tau \\
\text{const } \tau & \leq \tau' \\
\text{volatile } \tau & \leq \tau
\end{align*}
\]
Subtype compatibility: pointers

Note that for flexibility in dealing with arrays, we would like flexibility wrt pointers and arrays:

```c
int f(int *a, int bounds)
{
    ...
}
```

In particular, we would like to treat arrays as being compatible with pointers, otherwise:

```c
int f(int a[5])
{
    ...
}
```

Thus: $\tau_{\text{array}[n]} \leq \tau_{\text{ptr}}$.

Question: at what price flexibility?
Naturally, we would also like subtyping to be reflexive and transitive:

\[ \tau \leq \tau \quad \text{and} \quad \tau_1 \leq \tau_2 \leq \tau_3 \implies \tau_1 \leq \tau_3 \]
Assignable types

We can also use types to isolate assignable expressions— that is, \textit{lvalues}.

Any mutable scalar type can be assigned to, as can any pointer type:

\[
\Gamma \vdash \text{lval } \tau_m \quad \Gamma \vdash \text{lval } \tau_{\text{ptr}}
\]

We will use this relation in defining type checking for assignment expressions.
Type judgements

Now, we can define the form of type judgements:

\[ J ::= \Gamma \vdash e : \tau \quad \text{type judgements} \]

We say that a judgement \( \Gamma \vdash e : \tau \) is *valid* iff it can be deduced.

We write \( e : \tau \) iff \( \emptyset \vdash e : \tau \) is valid.

Note: A fringe benefit of type analysis is a “free-variable analysis”; if \( e : \tau \) then \( e \) contains no undeclared variables.
Typing expressions: basics

Our basic typing rules allow us to treat variables and integer and character literals:

\[
\Gamma(x) = \tau \\
\Gamma \vdash x : \tau
\]

\[
\Gamma \vdash n : \text{int} \quad \Gamma \vdash c : \text{char}
\]

Note that unbound variables will be untypable.
Typing expressions: restricting arithmetic opns

One of the first things we want to accomplish is to disallow \textit{pointer arithmetic}.

We also want to disallow mutation of \texttt{const} types.

Therefore, we restrict in/decrement to mutable scalar types:

\[
\frac{\Gamma \vdash e : \tau_m}{\Gamma \vdash e++ : \tau_m} \quad \frac{\Gamma \vdash e : \tau_m}{\Gamma \vdash ++e : \tau_m} \quad \frac{\Gamma \vdash e : \tau_m}{\Gamma \vdash e-- : \tau_m} \quad \frac{\Gamma \vdash e : \tau_m}{\Gamma \vdash --e : \tau_m}
\]
Typing expressions: restricting arithmetic opns

We also want a sensible result for negation:

\[
\frac{\Gamma \vdash e : \tau \quad \tau \leq \tau_i}{\Gamma \vdash -e : \tau_i} \quad \frac{\Gamma \vdash e : \tau_s}{\Gamma \vdash !e : \text{int}}
\]

Note that in an expression \(-e\), \(e\) may in fact be unsigned (but the result of the negation is not).
Typing expressions: restricting arithmetic opns

We will also restrict binary arithmetic operations to scalar types (what does it mean to add pointers?):

\[\Gamma \vdash e_1 : \tau' \quad \Gamma \vdash e_2 : \tau'' \quad \tau', \tau'' \leq \tau \]
\[\Gamma \vdash e_1 \ op \ e_2 : \tau\]

Note that the result type is an upper bound of the operand types—ensures compatibility.
Typing expressions: restricting assignment

*Assignment* is a very delicate operation. We want to ensure that only lval-
ues are assigned to, with compatible values.

Also want to ensure that combined arithmetic/assignment operations per-
mit only mutable scalars on LHS (no pointer arithmetic).

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_m & \quad \Gamma \vdash e_2 : \tau \quad \tau \leq \tau_m \\
& \quad \Gamma \vdash e_1 \text{op}_a e_2 : \tau_m \\
\Gamma \vdash e_1 : \tau & \quad \Gamma \vdash e_2 : \tau' \quad \vdash_{lval} \tau \quad \tau' \leq \tau \\
& \quad \Gamma \vdash e_1 = e_2 : \tau
\end{align*}
\]
Type checking function calls require that the type of actual parameters be compatible with declared type of formals:

\[
\begin{align*}
\Gamma \vdash e : \text{void} \rightarrow \tau \\
\Gamma \vdash e() : \tau \\
\Gamma \vdash e : \tau_1 \ast \cdots \ast \tau_n \rightarrow \tau \\
\forall 0 < j \leq n . (\Gamma \vdash e_j : \tau'_j) \land (\tau'_j \leq \tau_j) \\
\Gamma \vdash e(e_1, \ldots, e_n) : \tau
\end{align*}
\]
Typing expressions: arrays and references

Finally, we want to sensibly track the results of referencing, dereferencing, and array access:

\[
\frac{\Gamma \vdash e : \tau_{\text{ptr}}}{\Gamma \vdash \ast e : \tau}
\quad
\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \& e : \tau_{\text{ptr}}}
\quad
\frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau_{\text{ptr}}}{\Gamma \vdash e[n] : \tau}
\]
Checking statements

Since programs are built out of statements, we also define a pass/fail check on statements wrt a given environment, denoted $\Gamma \vdash_{ok} s$:

$$
\frac{
\Gamma \vdash e : \tau_s \quad \Gamma \vdash_{ok} s_1 \quad \Gamma \vdash_{ok} s_2
}{\Gamma \vdash_{ok} \text{if } (e) s_1 \text{ else } s_2}
\quad
\frac{
\Gamma \vdash e : \tau_s \quad \Gamma \vdash_{ok} s
}{\Gamma \vdash_{ok} \text{while } (e) s}
$$
Checking statements: blocks

Blocks need to be treated somewhat specially, because any declarations appearing at the beginning expand the environment:

\[
\forall s \in ss . \ (\Gamma; \text{bindings}(ds)) \vdash_{ok} s \\
\Gamma \vdash_{ok} \{ ds \ ss \}
\]
Checking statements: return values

A nice trick is to bind the expected return value for functions (more in a moment), and check this against the type of returned expressions:

\[
\frac{\Gamma(\text{ret}) = \text{void}}{\Gamma \vdash \text{ok} \ return;}
\]

\[
\frac{\Gamma \vdash e : \tau \quad \Gamma(\text{ret}) = \text{void}}{\Gamma \vdash \text{ok} \ return \ e;}
\]
Obtaining environments from programs

Now, we can extend our definition of bindings to function definitions:

\[
\text{bindings}(p \ (\tau_b \ x (ps) \ b)) = \begin{align*}
&\text{let } \Gamma_1 = \text{bindings}(p) \\
&\text{let } \Gamma_2 = \text{bindings}(ps) \\
&\text{let } \tau = \tau_1 \ast \cdots \ast \tau_n \\
&\quad \text{where } \Gamma_2 = (x_1 : \tau_1; \ldots, x_n : \tau_n) \\
&\text{let } \Gamma_3 = (\Gamma_1; \Gamma_2; x : \tau \rightarrow \tau_b) \\
&\text{if } ((\Gamma_3; \text{ret} : \tau_b) \vdash \text{ok } b) \text{ then } \Gamma_3 \text{ else fail}
\end{align*}
\]

\[
\text{bindings}(p \ (\tau_b \ast x (ps) \ b)) = \begin{align*}
&\text{let } \Gamma_1 = \text{bindings}(p) \\
&\text{let } \Gamma_2 = \text{bindings}(ps) \\
&\text{let } \tau = \tau_1 \ast \cdots \ast \tau_n \\
&\quad \text{where } \Gamma_2 = (x_1 : \tau_1; \ldots, x_n : \tau_n) \\
&\text{let } \Gamma_3 = (\Gamma_1; \Gamma_2; x : \tau \rightarrow \tau_b \text{ ptr}) \\
&\text{if } ((\Gamma_3; \text{ret} : \tau_b \text{ ptr}) \vdash \text{ok } b) \text{ then } \Gamma_3 \text{ else fail}
\end{align*}
\]
Obtaining environments from programs

To treat functions with empty parameter lists:

\[
\begin{align*}
\text{bindings}(p \ (\tau_b \ x \ () \ b)) &= \text{let } \Gamma_1 = \text{bindings}(p) \\
&\quad \text{let } \Gamma_2 = (\Gamma_1; x : \text{void} \to \tau_b) \\
&\quad \text{if } ((\Gamma'; \text{ret} : \tau_b) \vdash_{\text{ok}} b) \text{ then } \Gamma_2 \text{ else fail} \\
\text{bindings}(p \ (\tau_b \ * \ x \ () \ b)) &= \text{let } \Gamma_1 = \text{bindings}(p) \\
&\quad \text{let } \Gamma_2 = (\Gamma_1; x : \text{void} \to \tau_b \text{ptr}) \\
&\quad \text{if } ((\Gamma'; \text{ret} : \tau_b \text{ptr}) \vdash_{\text{ok}} b) \text{ then } \Gamma_2 \text{ else fail}
\end{align*}
\]

Note that bindings fails on programs that do not type check.
Static program type analysis

Type checking a program \( p \) succeeds iff \( \text{bindings}(p) \) does not fail.

In particular, the program shown at the beginning of lecture will not type check.