Midterm statistics

• Two bonus points.
  • How many control flow edges are there in an LLVM control flow graph with $N$ vertices?

• Mean: 69.5
• Median: 70
• Standard deviation: 15.8
• Most missed question: Suppose that we call `trace` with $n=3$. When `trace` exits, the value stored in `rcx` is equal to which C-language expression:
  (a) `(((long*)m + 12)`  (b) `2`  (c) `(((long*)m + 96)`  (d) `m[2][2]`
Is this grammar ambiguous?

\[
\begin{align*}
<S> & ::= \text{if } <E> \text{ then } <S> \text{ else } <S> \\
& \quad | \text{if } <E> \text{ then } <S> \\
& \quad | <\text{ident}> = <E> ; \\
& \quad | \text{while } <E> <S> \\
& \quad | \{ <B> \} \\
\end{align*}
\]

\[
\begin{align*}
<B> & ::= <B> <S> \\
& \quad | \epsilon \\
\end{align*}
\]

\[
\begin{align*}
<E> & ::= <\text{ident}> | \text{true} | \text{false} \\
<\text{ident}> & ::= x \mid y \mid z
\end{align*}
\]
Types
Well-formed types

• In languages with type definitions, need additional rules to define well-formed types
• Judgements take the form $H \vdash t$
  • $H$ is set of type names
  • $t$ is a type
  • $H \vdash t$ – “Assuming $H$ names well-formed types, $t$ is a well-formed type”
Well-formed types

- In languages with type definitions, need additional rules to define well-formed types
- Judgements take the form $H \vdash t$
  - $H$ is set of type names
  - $t$ is a type
  - $H \vdash t$ – “Assuming $H$ names well-formed types, $t$ is a well-formed type”

\[
\begin{array}{cccc}
\text{INT} & \text{BOOL} & \text{ARROW} & \text{NAMED} \\
H \vdash \text{int} & H \vdash \text{bool} & H \vdash t_1 \Rightarrow t_2 & H \vdash s \quad s \in H \\
\end{array}
\]
Well-formed types

• In languages with type definitions, need additional rules to define well-formed types
• Judgements take the form $H \vdash t$
  • $H$ is set of type names
  • $t$ is a type
  • $H \vdash t$ – “Assuming $H$ names well-formed types, $t$ is a well-formed type”

<table>
<thead>
<tr>
<th>INT</th>
<th>BOOL</th>
<th>ARROW</th>
<th>NAMED</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H \vdash$</td>
<td>$H \vdash$</td>
<td>$H \vdash t_1$ $H \vdash t_2$</td>
<td>$H \vdash s$ $s \in H$</td>
</tr>
<tr>
<td>int</td>
<td>bool</td>
<td>$H \vdash t_1 \rightarrow t_2$</td>
<td></td>
</tr>
</tbody>
</table>

• Note: also need to modify the typing rules & judgements. E.g.,

<table>
<thead>
<tr>
<th>FUN</th>
</tr>
</thead>
<tbody>
<tr>
<td>$H \vdash t_1$ $H, \Gamma { x \mapsto t_1 } \vdash e : t_2$</td>
</tr>
</tbody>
</table>

$H, \Gamma \vdash \text{fun}(x : t_1) \rightarrow e : t_1 \rightarrow t_2$
• In languages with statements, need additional rules to defined well-formed statements
• E.g., judgements may take the form $D; \Gamma; rt \vdash s$
  • $D$ maps type names to their definitions
  • $\Gamma$ is a type environment (variables $\rightarrow$ types)
  • $rt$ is a type
  • $D; \Gamma; rt \vdash s$ - “with type definitions $D$, assuming type environment $\Gamma$, $s$ is a valid statement within the context of a function that returns a value of type $rt$”
Statements

- In languages with statements, need additional rules to defined well-formed statements
- E.g., judgements may take the form $D; \Gamma; rt \vdash s$
  - $D$ maps type names to their definitions
  - $\Gamma$ is a type environment (variables $\rightarrow$ types)
  - $rt$ is a type
  - $D; \Gamma; rt \vdash s$ - “with type definitions $D$, assuming type environment $\Gamma$, $s$ is a valid statement within the context of a function that returns a value of type $rt$”

\[
\begin{align*}
\text{ASSIGN} & \quad \Gamma \vdash e : \Gamma(x) \\
& \quad \frac{}{D; \Gamma; rt \vdash x := e} \\
\text{RETURN} & \quad \Gamma \vdash e : rt \\
& \quad \frac{}{D; \Gamma; rt \vdash \text{return } e} \\
\text{DECL} & \quad \Gamma \vdash e : t \\
& \quad \frac{D; \Gamma; rt \vdash s_2}{D; \Gamma \{x \rightarrow t\}; rt \vdash s_2} \\
& \quad \frac{}{D; \Gamma; rt \vdash \text{var } x = e; s_2}
\end{align*}
\]
Extrinsic (sub)types

- **Extrinsic view** (Curry-style): a type is a property of a term. Think:
  - There is some set of values
    
    ```
    type value =
      | VInt of int
      | VBool of bool
    ```

    - Each type corresponds to a subset of values
      ```
      let typ_int = function
        | VInt _ -> true
        | _ -> false
      let typ_bool = function
        | VBool _ -> true
        | _ -> false
      ```

    - A term has type $t$ if it evaluates to a value of type $t$

- **Types may overlap.**
  ```
  let typ_nat = function
    | VInt x -> x >= 0
    | _ -> false
  ```
Subtyping

- Call \( s \) a **subtype** of type \( t \) if the values of type \( s \) is a subset of values of type \( t \)
- A subtyping judgement takes the form \( \vdash s <: t \)
  - “The type \( s \) is a subtype of \( t \)”
  - Liskov substitution principle: if \( s \) is a subtype of \( t \), then terms of type \( t \) can be replaced with terms of type \( s \) without breaking type safety.

<table>
<thead>
<tr>
<th>NatInt</th>
<th>Subsumption</th>
<th>Transitivity</th>
<th>Reflexivity</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \Gamma \vdash e : s \vdash s &lt;: t )</td>
<td>( \vdash t_1 &lt;: t_2 \vdash t_2 &lt;: t_3 )</td>
<td>( \vdash t &lt;: t )</td>
</tr>
<tr>
<td>( \vdash \text{nat} &lt;: \text{int} )</td>
<td>( \Gamma \vdash e : t )</td>
<td>( \vdash t_1 &lt;: t_3 )</td>
<td></td>
</tr>
</tbody>
</table>

- Subsumption: if \( s \) is a subtype of \( t \), then terms of type \( s \) can be used as if they were terms of type \( t \)
Casting

- **Upcasting**: Suppose $s <: t$ and $e$ has type $s$. May safety cast $e$ to type $t$.
  - Subsumption rule: upcast implicitly (C, Java, C++, ...)
    - Not necessarily a no-op
  - In OCaml: upcast $e$ to $t$ with $(e :> t)$ (important for type inference!)

- **Downcasting**: Suppose $s <: t$ and $e$ has type $t$. May not safety cast $e$ to type $s$.
  - **Checked downcasting**: check that downcasts are safe at runtime (Java, dynamic_cast in C++)
    - Type safe – throwing an exception is not the same as a type error
  - **Unchecked downcasting**: static_cast in C++
  - **No downcasting**: OCaml
Extending the subtype relation

**Tuple**

\[
\frac{\vdash t_1 <: s_1 \quad \ldots \quad \vdash t_n <: s_n}{\vdash t_1 \times \ldots \times t_n <: s_1 \times \ldots \times s_n}
\]

**List**

\[
\frac{\vdash s <: t}{\vdash s \text{ list <: } t \text{ list}}
\]

**Array**

\[
\frac{\vdash s <: t}{\vdash s \text{ array <: } t \text{ array}}
\]
Extending the subtype relation

**TUPLE**

\[
\frac{\vdash t_1 :: s_1 \quad \ldots \quad \vdash t_n :: s_n}{\vdash t_1 \times \ldots \times t_n :: s_1 \times \ldots \times s_n}
\]

\[
\frac{\vdash t_1 :: s_1 \quad \ldots \quad \vdash t_n :: s_n}{\vdash \{t_1, \ldots, t_n\} :: \{s_1, \ldots, s_n\}}
\]

**LIST**

\[
\frac{\vdash s :: t}{\vdash \text{List}\{s\} :: \text{List}\{t\}}
\]

**ARRAY**

\[
\frac{\vdash s :: t}{\vdash \text{Array}\{s\} :: \text{Array}\{t\}}
\]

- Array subtyping rule is **unsound** (Java!)
  Let \( \Gamma = [x \mapsto \text{nat array}] \)

\[
\begin{align*}
\frac{\vdash x :: \text{nat array} \quad \vdash \text{nat} :: \text{int}}{\vdash x :: \text{int array}} & \quad \frac{\vdash 0 :: \text{nat}}{\vdash -1 :: \text{int}}
\end{align*}
\]

\[
\Gamma \vdash x[0] := -1
\]
Immutable records

**RecordWidth**

\[ \frac{}{\vdash \{ \text{lab}_1 : s_1; \ldots; \text{lab}_m : s_m \} \ll \{ \text{lab}_1 : s_1; \ldots; \text{lab}_n : s_n \} \quad n < m} \]

**RecordDepth**

\[ \frac{}{\vdash s_1 \ll t_1 \quad \ldots \quad \vdash s_n \ll t_n} \]

\[ \vdash \{ \text{lab}_1 : s_n; \ldots; \text{lab}_m : s_n \} \ll \{ \text{lab}_1 : t_1; \ldots; \text{lab}_n : t_n \} \]
• Width subtyping is easy to compile
  • $s <: t$ means $\text{sizeof}(t) < \text{sizeof}(s)$, but field positions are the same ($e.lab$ compiled the same way, whether $e$ has type $s$ or type $t$)

• Depth subtyping is easy to compile
  • $s <: t$ means $\text{sizeof}(s) = \text{sizeof}(t)$, so again field positions are the same.

• How to compile records with width + depth subtyping?
• Width subtyping is easy to compile
  • $s <: t$ means $\text{sizeof}(t) < \text{sizeof}(s)$, but field positions are the same ($e.lab$ compiled the same way, whether $e$ has type $s$ or type $t$)

• Depth subtyping is easy to compile
  • $s <: t$ means $\text{sizeof}(s) = \text{sizeof}(t)$, so again field positions are the same.

• How to compile records with width + depth subtyping?
  • Add an indirection layer!
  • $\text{sizeof}(s*) = \text{sizeof}(t*)$
Function subtyping

In the function subtyping rule, we say that the argument type is *contravariant*, and the output type is *covariant*.

Some languages (Eiffel, Dart) have *covariant* argument subtyping. Not type-safe!