COS320: Compiling Techniques

Zak Kincaid

March 28, 2019

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?

```
    <S> ::= if <E> then <S> else <S>
          | if <E> then <S>
          | <ident> = <E>;
          | while <E> <S>
          | {<B>}
    <B> ::= <B><S>
          | ϵ
    <E> ::= <ident> | true | false
<ident> ::= x | y | z
```

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"

$$\textsc{Int}\quad \frac{}{H \vdash \texttt{int}} \qquad \textsc{Bool}\quad \frac{}{H \vdash \texttt{bool}} \qquad \textsc{Arrow}\quad \frac{H \vdash t_1 \qquad H \vdash t_2}{H \vdash t_1 \to t_2} \qquad \textsc{Named}\quad \frac{}{H \vdash s}\ s \in H$$

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"

$$\textsc{Int}\quad \frac{}{H \vdash \texttt{int}} \qquad \textsc{Bool}\quad \frac{}{H \vdash \texttt{bool}} \qquad \textsc{Arrow}\quad \frac{H \vdash t_1 \qquad H \vdash t_2}{H \vdash t_1 \to t_2} \qquad \textsc{Named}\quad \frac{}{H \vdash s}\; s \in H$$

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

$$\textsc{Fun}\quad \frac{H \vdash t_1 \qquad H, \Gamma\{x \mapsto t_1\} \vdash e : t_2}{H, \Gamma \vdash \textbf{fun}\ (x : t_1)\texttt{->}e : t_1 \to t_2}$$

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
  - Γ is a type environment (variables $\rightarrow$ types)
  - rt is a type
  - $D; \Gamma; rt \vdash s$ – "with type definitions D, assuming type environment Γ, 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
  - Γ is a type environment (variables $\rightarrow$ types)
  - rt is a type
  - $D; \Gamma; rt \vdash s$ – "with type definitions D, assuming type environment Γ, s is a valid statement within the context of a function that returns a value of type rt"

$$\textsc{Assign}\quad \frac{\Gamma \vdash e : \Gamma(x)}{D; \Gamma; rt \vdash x := e}$$

$$\textsc{Return}\quad \frac{\Gamma \vdash e : rt}{D; \Gamma; rt \vdash \textbf{return } e}$$

$$\textsc{Decl}\quad \frac{\Gamma \vdash e : t \qquad D; \Gamma\{x \mapsto t\}; rt \vdash s_2}{D; \Gamma; rt \vdash \textbf{var } x = e; s_2}$$

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 priciple: if s is a subtype of t, then terms of type t can be replaced with terms of type s without breaking type safety.

$$\textsc{NatInt}\quad \frac{}{\vdash \texttt{nat} <: \texttt{int}} \qquad \textsc{Subsumption}\quad \frac{\Gamma \vdash e : s \qquad \vdash s <: t}{\Gamma \vdash e : t} \qquad \textsc{Transitivity}\quad \frac{\vdash t_1 <: t_2 \qquad \vdash t_2 <: t_3}{\vdash t_1 <: t_3} \qquad \textsc{Reflexivity}\quad \frac{}{\vdash t <: t}$$

- 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

$$\textsc{Tuple}\quad \frac{\vdash t_1 <: s_1 \qquad \dots \qquad \vdash t_n <: s_n}{\vdash t_1 * \dots * t_n <: s_1 * \dots * s_n}$$

$$\textsc{List}\quad \frac{\vdash s <: t}{\vdash s\ \texttt{list} <: t\ \texttt{list}}$$

$$\textsc{Array}\quad \frac{\vdash s <: t}{\vdash s\ \texttt{array} <: t\ \texttt{array}}$$

Extending the subtype relation

$$\textsc{Tuple}\;\frac{\vdash t_1 <: s_1 \qquad \ldots \qquad \vdash t_n <: s_n}{\vdash t_1 * \ldots * t_n <: s_1 * \ldots * s_n} \qquad \textsc{List}\;\frac{\vdash s <: t}{\vdash s\ \texttt{list} <: t\ \texttt{list}} \qquad \textsc{Array}\;\frac{\vdash s <: t}{\vdash s\ \texttt{array} <: t\ \texttt{array}}$$

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

$$\textsc{Assn}\;\dfrac{\textsc{Sub}\;\dfrac{\textsc{Var}\;\dfrac{}{\Gamma \vdash x : \texttt{nat array}} \quad \textsc{Array}\;\dfrac{\textsc{NatInt}\;\dfrac{}{\texttt{nat} <: \texttt{int}}}{\texttt{nat array} <: \texttt{int array}}}{\Gamma \vdash x : \texttt{int array}} \qquad \textsc{Nat}\;\dfrac{}{\Gamma \vdash 0 : \texttt{nat}} \qquad \textsc{Int}\;\dfrac{}{\Gamma \vdash -1 : \texttt{int}}}{\Gamma \vdash x[0] := -1}$$

Immutable records

$$\textsc{RecordWidth}\quad \frac{}{\vdash \{\textit{lab}_1 : s_1; \ldots; \textit{lab}_m : s_m\} <: \{\textit{lab}_1 : s_1; \ldots; \textit{lab}_n : s_n\}}\ n < m$$

$$\textsc{RecordDepth}\quad \frac{\vdash s_1 <: t_1 \qquad \ldots \qquad \vdash s_n <: t_n}{\vdash \{\textit{lab}_1 : s_n; \ldots; \textit{lab}_m : s_n\} <: \{\textit{lab}_1 : t_1; \ldots; \textit{lab}_n : t_n\}}$$

- Width subtyping is easy to compile
  - $s <: t$ means $\texttt{sizeof}(t) < \texttt{sizeof}(s)$, 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 $\texttt{sizeof}(s) = \texttt{sizeof}(t)$, so again field positions are the same.
- How to comple records with width + depth subtyping?

- Width subtyping is easy to compile
  - $s <: t$ means $\texttt{sizeof}(t) < \texttt{sizeof}(s)$, 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 $\texttt{sizeof}(s) = \texttt{sizeof}(t)$, so again field positions are the same.
- How to comple records with width + depth subtyping?
  - Add an indirection layer!
  - $\texttt{sizeof}(s*) = \texttt{sizeof}(t*)$

Function subtyping

$$\textsc{Fun}\;\frac{\vdash s_1 <: t_1 \qquad \vdash t_2 <: s_2}{\vdash t_1 \to t_2 <: s_1 \to s_2}$$

- 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!