Monomorphic Type Systems

Meeting 26, CSCI 5535, Spring 2010

Homework Stats

- HW0 (out of 20)
  - mean 18.6, stddev 1.9, median 20, 3rd quartile 20, max 20
- HW1 (out of 50)
  - mean 46.5, stddev 5.6, median 49, 3rd quartile 50, max 50
- HW2 (out of 50)
  - mean 43.0, stddev 8.1, median 45, 3rd quartile 48, max 50
- HW3 (out of 50)
  - mean 41.3, stddev 10.8, median 44, 3rd quartile 50, max 50
- HW4 (out of 50)
  - mean 38.0, stddev 12.7, median 39, 3rd quartile 48, max 50
- HW5 (out of 48)
  - mean 39.7, stddev 7.0, median 41, 3rd quartile 44, max 48

Review of the Static Semantics of the Simply-Typed Lambda Calculus

Typing Judgments

- A common form of typing judgment:
  \[ \Gamma \vdash e : \tau \] (\( e \) is an expression and \( \tau \) is a type)
- \( \Gamma \) (Gamma) is a set of type assignments for the free variables of \( e \)
  - Defined by the grammar \( \Gamma ::= \emptyset \mid \Gamma, x : \tau \)
  - “Assuming type assignments for variables in \( \Gamma \), expression \( e \) has type \( \tau \).”

Simply-Typed Lambda Calculus

- Syntax:
  - Terms: \( e ::= x \mid \lambda x : \tau . e \mid e_1 e_2 \mid n \mid e_1 + e_2 \mid \text{iszero } e \mid \text{true} \mid \text{false} \mid \text{not } e \mid \text{if } e \text{ then } e_1 \text{ else } e_3 \)
  - Types: \( \tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2 \)

- \( \tau_1 \rightarrow \tau_2 \) is the function type
- \( \rightarrow \) associates to the right
- This language is also called \( F_1 \)

Static Semantics of \( F_1 \)

- Function rules
  \[
  \begin{align*}
  e : \tau & \in \Gamma \\
  \Gamma, x : \tau \vdash e : \tau' \\
  \Gamma, \lambda x : \tau . e : \tau' \\
  \Gamma, e_1 : \tau_1 \rightarrow \tau, e_2 : \tau_2 \vdash e_1 e_2 : \tau
  \end{align*}
  \]
More Static Semantics of $F_1$

- **Base type rules**
  \[
  \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}
  \]
  \[
  \frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{true} : \text{bool}}
  \]
  \[
  \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}
  \]

Type Checking in $F_1$

- **Type checking is easy** because
  - Typing rules are *syntax directed*
  - Typing rules are *compositional*
  - All local variables are annotated with types
- In fact, *type inference is also easy* for $F_1$
- Without type annotations an expression may have *no unique type*

\[
\frac{}{\vdash \lambda x : \tau. e : \tau \rightarrow \tau}
\]

Formalizing a Language

1. **Syntax**
   - Of expressions (programs), of types
   - Issues of binding and scoping
2. **Static semantics (typing rules)**
   - Define the typing judgment and its derivation rules
3. **Dynamic Semantics (e.g., operational)**
   - Define the evaluation judgment and its derivation rules
4. **Type soundness**
   - Relates the static and dynamic semantics
   - State and prove the *soundness theorem*

Operational Semantics of $F_1$

- **Judgment:**
  \[
  e \Downarrow v
  \]
- **Values:**
  \[
  v ::= n \mid \text{true} \mid \text{false} \mid \lambda x : \tau. e
  \]
- **The evaluation rules**...
  - **Audience participation time:** give me an evaluation rule.
**Operational Semantics of F₁**

**Call-by-value (sample)**

\[
\begin{align*}
\lambda x: \tau. e & \downarrow \lambda x: \tau. e \\
\text{e}_1 \downarrow \lambda x: \tau. e' & \quad \text{e}_2 \downarrow \text{v}_2 \quad e_1[e_2/x] \downarrow \text{v} \\
\text{e}_1 \downarrow v & \quad \text{e}_2 \downarrow v \\
\frac{}{n \downarrow n} & \quad e_1 + e_2 \downarrow n \\
\frac{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \downarrow v}{e_1 \downarrow \text{true} \quad e_2 \downarrow v} & \quad \frac{e_1 \downarrow \text{false} \quad e_3 \downarrow v}{e_1 \downarrow \text{false} \quad e_2 \downarrow e_3 \downarrow v} \\
\end{align*}
\]

Evaluation undefined for ill-typed programs!

---

**Type Soundness for F₁**

- **Theorem:** If \( \vdash e : \tau \) and \( e \downarrow v \) then \( \vdash v : \tau \)
  - Also called, subject reduction theorem, type preservation theorem
  - This is one of the most important sorts of theorems in PL
  - Whenever you make up a new safe language you are expected to prove this
  - Examples: Vault, TAL, CCured, ...

**How Might We Prove It?**

If \( T :: \vdash e : \tau \) and \( E :: e \downarrow v \) then \( \vdash v : \tau \)
How Might We Prove It?

If \( T \vdash e : \tau \) and \( E \vdash e \Downarrow v \) then \( \Downarrow v : \tau \)

Proof Approaches to Type Safety

- By induction on \( e \)?
  - Won't work because \([v_2/x]e'_1\) in the eval of \( e_1 e_2 \)
  - Same problem with induction on \( T \)
- By induction on \( \tau \)?
  - Won't work because \( e_1 \) has a "bigger" type than \( e_1 e_2 \)
- By induction on \( E \)?
  - To address the issue of \([v_2/x]e'_1\)
  - This is it!

Type Soundness Proof

Consider the function application case

\[
E :: E_1 :: e_1 ::= \lambda x : \tau_2, e'_1 :: e_2 :: e_2 :: e_3 :: [v_2/x]e'_1 :: v \quad e_1 e_2 :: v
\]

...
Significance of Type Soundness

• The theorem says that the result of an evaluation has the same type as the initial expression
• The theorem does not say that
  - The evaluation never gets stuck (e.g., trying to apply a non-function, to add non-integers, etc.), nor that
  - The evaluation terminates
• Even though both of the above facts are true of $F_1$
• What formal system of semantics do we use to reason about programs that might not terminate?

Small-Step Contextual Semantics for $F_1$

• We define redexes
  $$r ::= n_1 + n_2 \mid \text{if} \ b \ \text{then} \ e_1 \ \text{else} \ e_2 \mid (\lambda x: \tau. e_1) \ v_2$$
• and contexts
  $$H ::= H_1 + e_2 \mid n_1 + H_2 \mid \text{if} \ H \ \text{then} \ e_1 \ \text{else} \ e_2 \mid H_1 e_2 \mid (\lambda x: \tau. e_1) H_2 \mid \bullet$$
• and local reduction rules
  $$n_1 + n_2 \rightarrow n_1 \text{ plus } n_2$$
  $$\text{if true then } e_1 \ \text{else} \ e_2 \rightarrow e_1$$
  $$\text{if false then } e_1 \ \text{else} \ e_2 \rightarrow e_2$$
  $$\frac{}{v_2/x} e_2$$
• and one global reduction rule
  $$H[r] \rightarrow H[e] \text{ if } r \rightarrow e$$

Decomposition Lemmas for $F_1$

- If $\vdash e : \tau$ and $e$ is not a (final) value then there exist (unique) $H$ and $r$ such that $e = H[r]$
- Any well-typed expression can be decomposed
- Any well-typed non-value can make progress
- Furthermore, there exists $c$ such that $\vdash r : c$
  - The redex is closed and well typed
- Furthermore, there exists $e'$ such that $r \rightarrow e'$ and $\vdash e' : \tau$
  - Local reduction is type preserving
- Furthermore, for any $e'$, $\vdash e' : \tau$ implies $\vdash H[e'] : \tau$
  - The expression preserves its type if we replace the redex with an expression of same type

Type Safety of $F_1$

• Type preservation theorem
  - If $\vdash e : \tau$ and $e \rightarrow e'$ then $\vdash e' : \tau$
  - Follows from the decomposition lemma
• Progress theorem
  - If $\vdash e : \tau$ and $e$ is not a value then there exists $e'$ such that $e$ can make progress: $e \rightarrow e'$
• Progress theorem says that execution can make progress on a well typed expression
• From type preservation we know the execution of well typed expressions never gets stuck
  - This is a (very) common way to state and prove type safety of a language
What’s Next?
• We’ve got the basic simply-typed monomorphic lambda calculus
• Now let’s make it more complicated ...
• By adding features!

Products: Syntax and Static Semantics
• Extend the syntax with (binary) tuples
  \[ e ::= ... \mid (e_1, e_2) \mid \text{fst } e \mid \text{snd } e \]
  \[ \tau ::= ... \mid \tau_1 \times \tau_2 \]
  - This language is sometimes called F₁
• Same typing judgment
  \[ \Gamma \vdash e : \tau \]
  \[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \]
  \[ \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \]
  \[ \Gamma \vdash e : \tau_1 \times \tau_2 \quad \Gamma \vdash e : \tau_1 \times \tau_2 \]
  \[ \Gamma \vdash \text{fst } e : \tau_1 \quad \Gamma \vdash \text{snd } e : \tau_2 \]

Products: Dynamic Sem. and Soundness
• New form of values: \[ v ::= ... \mid (v_1, v_2) \]
• New (big step) evaluation rules:
  \[ e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \]
  \[ (e_1, e_2) \Downarrow (v_1, v_2) \]
  \[ e \Downarrow (v_1, v_2) \quad e \Downarrow (v_1, v_2) \]
  \[ \text{fst } e \Downarrow v_1 \quad \text{snd } e \Downarrow v_2 \]
• New contexts:
  \[ H ::= ... \mid (H_1, e_2) \mid (v_1, H_2) \mid \text{fst } H \mid \text{snd } H \]
• New redexes:
  \[ \text{fst } (v_1, v_2) \rightarrow v_1 \]
  \[ \text{snd } (v_1, v_2) \rightarrow v_2 \]
• Type soundness holds just as before

General PL Feature Plan
• The general plan for language feature design
  - You invent a new feature (tuples)
  - You add it to the lambda calculus
  - You invent typing rules and opsem rules
  - You extend the basic /proof of type safety
  - You declare moral victory, and milling throngs of cheering admirers wait to carry you on their shoulders to be knighted by the Queen, etc.

Two new features ...

Records
• Records are like tuples with labels
  - New form of expressions
    \[ e ::= ... \mid (L_1 = e_1, ..., L_n = e_n) \mid e \cdot L \]
  - New form of values
    \[ v ::= (L_1 = v_1, ..., L_n = v_n) \]
  - New form of types
    \[ \tau ::= ... \mid (L_1 : \tau_1, ..., L_n : \tau_n) \]
  - ... follows the model of F₁
Sums

- We need **disjoint union types** of the form:
  - either an int or a float
  - either 0 or a pointer
  - either a (binary tree node with two children) or a (leaf)

- New expressions and types

  e ::= ... | injl e | injr e | case e of injl x → e₁ | injr y → e₂

  τ ::= ... | τ₁ + τ₂

- A value of type τ₁ + τ₂ is either a τ₁ or a τ₂
- Like union in C or Pascal, but safe
- distinguishing between components is under compiler control
- case is a binding operator (like "let"): x is bound in e₁ and y is bound in e₂ (like OCaml’s "match ... with")

Examples with Sums

- Consider the type **unit** with a single element called * or ()
- The type **integer option** defined as "unit + int"
  - Useful for optional arguments or return values
  - No argument: injl * (OCaml’s "None")
  - Argument is 5: injr 5 (OCaml’s "Some(5)"

  - To use the argument you must test the kind of argument
  - case arg of injl x ⇒ "no_arg_case" | injr y ⇒ "...y..."
  - injl and injr are tags and case is tag checking

- bool is the union type "unit + unit"
  - true is injl *
  - false is injr *
  - if e then e₁ else e₂ is case e of injl x ⇒ e₁ | injr y ⇒ e₂

Static and Dynamic Semantics for Records and Sums

- Try it on paper and then volunteer to come on down!
  - New typing rules for Γ ⊢ e : τ
  - New values v ::= ... | injl v | injr v
  - New evaluation rules for e ↓ v
  - (Extra) new contexts H ::= ...
  - (Extra) new redexes r ::= ...
  - (Extra) new local reduction rules r → e

Dynamic Semantics of Sums

- New values v ::= ... | injl v | injr v
- New evaluation rules

  \[
  \begin{align*}
  e & \downarrow v \\
  \text{injl } e & \downarrow \text{injl } v \\
  \text{injr } e & \downarrow \text{injr } v \\
  e & \downarrow \text{injl } v \quad \text{[v/x]} e₁ \downarrow v' \\
  \text{case } e & \text{ of injl } x ⇒ e₁ | \text{injr } y ⇒ e₂ \downarrow v' \\
  e & \downarrow \text{injr } v \quad \text{[v/y]} e₂ \downarrow v' \\
  \text{case } e & \text{ of injl } x ⇒ e₁ | \text{injr } y ⇒ e₂ \downarrow v'
  \end{align*}
\]

Type Soundness for F₁⁺

- Type soundness **still holds**
- No way to use a τ₁ + τ₂ inappropriately
- The key is that the **only way to use a τ₁ + τ₂ is with case**, which ensures that you are not using a τ₁ as a τ₂
- In C or Pascal checking the tag is the responsibility of the programmer!
- Unsafe
For Next Time

- Read Wright and Felleisen paper
  - that you might not have read for today 😊
- Work on projects