Monomorphic Type Systems
Meeting 19, CSCI 5535, Spring 2009

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 \( \Gamma ::= \epsilon | \Gamma, x : \tau \)
  - "Assuming type assignments for variables in \( \Gamma \), expression \( e \) has type \( \tau \)."

Simply-Typed Lambda Calculus
- Syntax:
  * Terms: \( e ::= x | \lambda x : \tau. e | e_1 e_2 | n | e_1 + e_2 | \text{iszero} e | \text{true} | \text{false} | \text{not} e | \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \)
  * Types: \( \tau ::= \text{int} | \text{bool} | \tau_1 \to \tau_2 \)
  - \( \tau_1 \to \tau_2 \) is the function type
  - \( \to \) associates to the right
  - This language is also called F

Static Semantics of \( F_1 \)
- Function rules
  \[
  \begin{align*}
  &x : \tau \in \Gamma \quad \Gamma, x : \tau \vdash e : \tau' \\
  &\Gamma \vdash x : \tau \\
  &\Gamma \vdash \lambda x : \tau. e : \tau \to \tau'
  
  &\Gamma \vdash e_1 : \tau_2 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \\
  &\Gamma \vdash e_1 e_2 : \tau
  \end{align*}
  \]

More Static Semantics of \( F_1 \)
- Base type rules
  - \( \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \)
  - \( \Gamma \vdash n : \text{int} \quad \Gamma \vdash e_1 + e_2 : \text{int} \)
  - \( \Gamma \vdash e : \text{bool} \quad \Gamma \vdash \text{true} : \text{bool} \quad \Gamma \vdash \text{not } e : \text{bool} \)
  - \( \Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \\
  &\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau \)
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.

```
⊢ λx. x : int → int
⊢ λx. x : bool → bool
```

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:
  
  \[ v ::= n \mid true \mid false \mid \lambda x : \tau. e \]

- The evaluation rules ...
  - Audience participation time: give me an evaluation rule.
Operational Semantics of F₁:
Call-by-value (sample)

\[ \begin{align*}
\lambda x : \tau. e &\downarrow \lambda x : \tau. e \\
e_1 \downarrow \lambda x : \tau. e', e_2 \vdash \mathcal{K}_2 \mathcal{K}(\lambda x : \tau. e_1)^r \downarrow v \\
e_1 &\downarrow v_1 \\
e_2 &\downarrow v_2 \\
n &\downarrow n \\
 &\downarrow n + n_2 \\
 &\downarrow e_1 + e_2 + n \\
&\downarrow \text{true if } e_1 \text{ else } e_2 + v \\
&\downarrow \text{false if } e_1 \text{ else } e_3 + v
\end{align*} \]

Type Soundness for F₁

- Theorem:
  \[ \text{If } \vdash e : \tau \text{ and } e \Downarrow v \text{ then } \vdash v : \tau \]
  - 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?

\[ \begin{align*}
\text{If } T &\vdash e : \tau \text{ and } E \vdash e \Downarrow v \text{ then } \vdash v : \tau
\end{align*} \]
How Might We Prove It?

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

Proof Approaches to Type Safety

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

- 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 $e_2$
- 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
  $E : E_1 :: e_1 \Downarrow \lambda x : \tau_2. e_1' E_2 :: e_2 : \tau_2 E_3 :: [v_2/x]e_1' \Downarrow v$

  $\vdash e_1 : \tau$ $\vdash e_2 : \tau$ $\vdash [v_2/x]e_1' : \tau$ $\vdash v : \tau$

  $T_1 : \vdash e_1 : \tau_2 \rightarrow \tau$ $T_2 : \vdash e_2 : \tau_2$

  and by inversion on $T$

  $\vdash e_1 : \tau_2$ $\vdash e_2 : \tau_2$

  By i.h. on $E_1$ with $T_1$, $\lambda x : \tau_2. e_1' : \tau_2 \rightarrow \tau$

  By i.h. on $E_2$ with $T_2$, $v_2 : \tau_2$

  $T_3 : \vdash \ldots$

Significance of Type Soundness

• The theorem says that the result of an evaluation has the same type as the initial expression $e$

• 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?
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 $F_1$.
- We need a small-step semantics to prove that the execution never gets stuck.
- I Assert: the execution always terminates in $F_1$.
- When does the base lambda calculus ever not terminate?

Small-Step Contextual Semantics for $F_1$
- We define redexes:
  
  \[
  r ::= n_1 + n_2 \mid \text{if } b \text{ then } e_1 \text{ else } e_2 \mid (\lambda x : \tau . e_1) v_2 \n  \]
- and contexts:
  
  \[
  H ::= H_1 + e_2 \mid n_1 + H_2 \mid \text{if } H \text{ then } e_1 \text{ else } e_2 \mid (\lambda x : \tau . e_1) H_2 \mid \ast \n  \]
- and local reduction rules:
  
  \[
  n_1 + n_2 \rightarrow n_1 \text{ plus } n_2 \n  \]
  
  \[
  \text{if true then } e_1 \text{ else } e_2 \rightarrow e_1 \n  \]
  
  \[
  \text{if false then } e_1 \text{ else } e_2 \rightarrow e_2 \n  \]
  
  \[
  (\lambda x : \tau . e_1) v_2 \rightarrow [v_2/x]e_1 \n  \]
- and one global reduction rule:
  
  \[
  H[r] \rightarrow H[e] \text{ if } r \rightarrow e \n  \]

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 $\varepsilon$ such that $\vdash \varepsilon : \iota$.
- The redex is closed and well-typed.
- Furthermore, there exists $\varepsilon'$ such that $r \rightarrow \varepsilon'$ and $\vdash \varepsilon' : \iota$.
- Local reduction is type preserving.
- Furthermore, for any $\varepsilon', \vdash \varepsilon' : \iota$ implies $\vdash H[\varepsilon'] : \tau$.
- 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 $e \rightarrow e'$.
  - 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.

Type Safety

Preservation

Progress

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
  \[ e ::= ... | (e_1, e_2) | \text{fst } e | \text{snd } e \]

- The syntax is sometimes called $F_1^\times$

- Same typing judgment
  \[ \Gamma \vdash e : \tau \]

- New expressions
  \[ \Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2 \]

- New contexts
  \[ \Gamma \vdash \text{fst } e : \tau_1 \]

- New values
  \[ \Gamma \vdash \text{snd } e : \tau_2 \]

Products: Dynamic Sem. and Soundness

- New form of values: \[ v ::= ... | (v_1, v_2) \]

- New (big step) evaluation rules:

- New contexts:

- New redexes:

- 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.

Records

- Records are like tuples with labels

- New form of expressions
  \[ e ::= ... | (L_1 = e_1, ..., L_n = e_n) | e.L \]

- New form of values
  \[ v ::= (L_1 = v_1, ..., L_n = v_n) \]

- New form of types
  \[ \tau ::= ... | L_1 : \tau_1, ..., L_n : \tau_n \]

- ... follows the model of $F_1^\times$

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
  \[ e ::= ... | \text{injl } e | \text{injr } e | \text{case } e \text{ of injl } x \rightarrow e_1 | \text{injr } y \rightarrow e_2 \]

- A value of type $\tau_1 + \tau_2$ is either a $\tau_1$ or a $\tau_2$

- 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 $e_1$ and $y$ is bound in $e_2$ (like OCaml's "match ... with")
Examples with Sums

• Consider the type `unit` with a single element called "*: 0 ()

• 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)"

• 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 \Rightarrow \text{"no_arg_case"} | injr y \Rightarrow \text{"...y..."}

  • `bool` is the union type "unit + unit":
    - true is `injl *`
    - false is `injr *`
    - `if e then e_1 else e_2` is case `e` of `injl x \Rightarrow e_1 | injr y \Rightarrow e_2`

Static and Dynamic Semantics for Records and Sums

• Try it on paper and then volunteer to come on down!
  - New typing rules for \( \Gamma \vdash e : \tau \)
  - New values \( v ::= \ldots | \injl v | \injr v \)
  - New evaluation rules for \( e \Downarrow v \)
    - `injl e \Downarrow \injl v`
    - `injr e \Downarrow \injr v`
    - `e \Downarrow \injl v [v/x]e_1 \Downarrow v'`
    - `e \Downarrow \injr v [v/y]e_r \Downarrow v'`
  - (Extra) new redexes \( n ::= \ldots \)
  - (Extra) new local reduction rules \( r \rightarrow e \)

Static Semantics of Sums

• New typing rules
  \[
  \begin{align*}
  \Gamma \vdash e : \tau_1 & \quad \Gamma \vdash e : \tau_2 \\
  \Gamma \vdash \injl e : \tau_1 + \tau_2 & \quad \Gamma \vdash \injr e : \tau_1 + \tau_2 \\
  \Gamma \vdash e_1 : \tau_1 + \tau_2 & \quad \Gamma \vdash x : \tau_1 & \quad \Gamma \vdash y : \tau_2 & \quad \Gamma \vdash e : \tau \\
  \Gamma \vdash \text{case } e_1 \text{ of } \injl x \Rightarrow e_1 | \injr y \Rightarrow e_r : \tau
  \end{align*}
  \]

  • Types are not unique anymore
    - `injl : int + bool`
    - `inj 1 : int + (int \rightarrow int)`
    - this complicates type checking, but it is still doable

Dynamic Semantics of Sums

• New values \( v ::= \ldots | \injl v | \injr v \)
• New evaluation rules
  \[
  \begin{align*}
  e \Downarrow v & \quad e \Downarrow v \\
  \injl e \Downarrow \injl v & \quad \injr e \Downarrow \injr v \\
  e \Downarrow \injl v [v/x]e_1 \Downarrow v' & \quad e \Downarrow \injr v [v/y]e_r \Downarrow v' \\
  \text{case } e \text{ of } \injl x \Rightarrow e_1 | \injr y \Rightarrow e_r : v
  \end{align*}
  \]

Type Soundness for \( F_1^+ \)

• Type soundness still holds
• No way to use a \( \tau_1 + \tau_2 \) inappropriately

  • The key is that the only way to use a \( \tau_1 + \tau_2 \) is with case, which ensures that you are not using a \( \tau_1 \) as a \( \tau_2 \)

  • 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, status updates