CS-XXX: Graduate Programming Languages

Lecture 11 — STLC Extensions and Related Topics

Dan Grossman
2012

Review

$$\begin{array}{lcl} e & ::= & \lambda x.\ e \mid x \mid e\ e \mid c \\ v & ::= & \lambda x.\ e \mid c \end{array} \qquad \begin{array}{lcl} \tau & ::= & \mathsf{int} \mid \tau \to \tau \\ \Gamma & ::= & \cdot \mid \Gamma, x : \tau \end{array}$$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

$e[e'/x]$: capture-avoiding substitution of e' for free x in e

$$\frac{}{\Gamma \vdash c : \mathsf{int}} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.\ e : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\ e_2 : \tau_1}$$

Preservation: If $\cdot \vdash e : \tau$ and $e \to e'$, then $\cdot \vdash e' : \tau$.
Progress: If $\cdot \vdash e : \tau$, then e is a value or $\exists\, e'$ such that $e \to e'$.

Adding Stuff

Time to use STLC as a foundation for understanding other common language constructs

We will add things via a *principled methodology* thanks to a *proper education*

- Extend the syntax
- Extend the operational semantics
  - Derived forms (syntactic sugar), or
  - Direct semantics
- Extend the type system
- Extend soundness proof (new stuck states, proof cases)

In fact, extensions that add new types have even more structure

Let bindings (CBV)

$$e ::= \ldots \mid \textbf{let } x = e_1 \textbf{ in } e_2$$

$$\frac{e_1 \to e_1'}{\textbf{let } x = e_1 \textbf{ in } e_2 \to \textbf{let } x = e_1' \textbf{ in } e_2} \qquad \frac{}{\textbf{let } x = v \textbf{ in } e \to e[v/x]}$$

$$\frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma, x : \tau' \vdash e_2 : \tau}{\Gamma \vdash \textbf{let } x = e_1 \textbf{ in } e_2 : \tau}$$

(Also need to extend definition of substitution...)

Progress: If e is a let, 1 of the 2 new rules apply (using induction)

Preservation: Uses Substitution Lemma

Substitution Lemma: Uses Weakening and Exchange

Derived forms

let seems just like λ, so can make it a derived form

- **let** $x = e_1$ **in** e_2 "a macro" / "desugars to" $(\lambda x.\ e_2)\ e_1$
- A "derived form"

(Harder if λ needs explicit type)

Or just define the semantics to replace let with λ:

$$\frac{}{\textbf{let } x = e_1 \textbf{ in } e_2 \rightarrow (\lambda x.\ e_2)\ e_1}$$

These 3 semantics are *different* in the state-sequence sense $(e_1 \rightarrow e_2 \rightarrow \ldots \rightarrow e_n)$

- But (totally) *equivalent* and you could prove it (not hard)

Note: ML type-checks let and λ differently (later topic)
Note: Don't desugar early if it hurts error messages!

Booleans and Conditionals

$$
\begin{array}{rcl}
e & ::= & \ldots \mid \textbf{true} \mid \textbf{false} \mid \textbf{if}\ e_1\ e_2\ e_3 \\
v & ::= & \ldots \mid \textbf{true} \mid \textbf{false} \\
\tau & ::= & \ldots \mid \textbf{bool}
\end{array}
$$

$$\frac{e_1 \rightarrow e_1'}{\textbf{if}\ e_1\ e_2\ e_3 \rightarrow \textbf{if}\ e_1'\ e_2\ e_3}$$

$$\frac{}{\textbf{if true}\ e_2\ e_3 \rightarrow e_2} \qquad \frac{}{\textbf{if false}\ e_2\ e_3 \rightarrow e_3}$$

$$\frac{\Gamma \vdash e_1 : \textbf{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textbf{if}\ e_1\ e_2\ e_3 : \tau}$$

$$\frac{}{\Gamma \vdash \textbf{true} : \textbf{bool}} \qquad \frac{}{\Gamma \vdash \textbf{false} : \textbf{bool}}$$

Also extend definition of substitution (will stop writing that)...

Notes: CBN, new Canonical Forms case, all lemma cases easy

Pairs (CBV, left-right)

$$\begin{array}{rcl} e & ::= & \ldots \mid (e, e) \mid e.1 \mid e.2 \\ v & ::= & \ldots \mid (v, v) \\ \tau & ::= & \ldots \mid \tau * \tau \end{array}$$

$$\frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)} \qquad \frac{e_2 \to e_2'}{(v_1, e_2) \to (v_1, e_2')}$$

$$\frac{e \to e'}{e.1 \to e'.1} \qquad \frac{e \to e'}{e.2 \to e'.2}$$

$$\frac{}{(v_1, v_2).1 \to v_1} \qquad \frac{}{(v_1, v_2).2 \to v_2}$$

Small-step can be a pain

- Large-step needs only 3 rules
- Will learn more concise notation later (evaluation contexts)

Pairs continued

$$\frac{\Gamma \vdash e_1 : \tau_1 \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 * \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.1 : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 * \tau_2}{\Gamma \vdash e.2 : \tau_2}$$

Canonical Forms: If $\cdot \vdash v : \tau_1 * \tau_2$, then v has the form (v_1, v_2)

Progress: New cases using Canonical Forms are $v.1$ and $v.2$

Preservation: For primitive reductions, inversion gives the result *directly*

Records

Records are like n-ary tuples except with *named fields*

- Field names are *not* variables; they do *not* α-convert

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \{l_1 = e_1; \ldots ; l_n = e_n\} \mid e.l \\
v & ::= & \ldots \mid \{l_1 = v_1; \ldots ; l_n = v_n\} \\
\tau & ::= & \ldots \mid \{l_1 : \tau_1; \ldots ; l_n : \tau_n\}
\end{array}
$$

$$
\frac{e_i \to e_i'}{\begin{array}{c}\{l_1{=}v_1, \ldots, l_{i-1}{=}v_{i-1}, l_i{=}e_i, \ldots, l_n{=}e_n\} \\ \to \{l_1{=}v_1, \ldots, l_{i-1}{=}v_{i-1}, l_i{=}e_i', \ldots, l_n{=}e_n\}\end{array}} \qquad \frac{e \to e'}{e.l \to e'.l}
$$

$$
\frac{1 \leq i \leq n}{\{l_1 = v_1, \ldots, l_n = v_n\}.l_i \to v_i}
$$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \qquad \ldots \qquad \Gamma \vdash e_n : \tau_n \qquad \text{labels distinct}}{\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}}
$$

$$
\frac{\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \qquad 1 \leq i \leq n}{\Gamma \vdash e.l_i : \tau_i}
$$

Records continued

Should we be allowed to reorder fields?

- $\cdot \vdash \{l_1 = 42; l_2 = \textbf{true}\} : \{l_2 : \textbf{bool}; l_1 : \textbf{int}\}$??
- Really a question about, "when are two types equal?"

Nothing wrong with this from a type-safety perspective, yet many languages disallow it

- Reasons: Implementation efficiency, type inference

Return to this topic when we study *subtyping*

Sums

What about ML-style datatypes:

```
type t = A | B of int | C of int * t
```

1. Tagged variants (i.e., discriminated unions)
2. Recursive types
3. Type constructors (e.g., `type 'a mylist = ...`)
4. Named types

For now, just model (1) with (anonymous) sum types

- (2) is in a later lecture, (3) is straightforward, and (4) we'll discuss informally

Sums syntax and overview

$$
\begin{array}{lcl}
e & ::= & \ldots \mid \mathbf{A}(e) \mid \mathbf{B}(e) \mid \textbf{match } e \textbf{ with } \mathbf{A}x.\ e \mid \mathbf{B}x.\ e \\
v & ::= & \ldots \mid \mathbf{A}(v) \mid \mathbf{B}(v) \\
\tau & ::= & \ldots \mid \tau_1 + \tau_2
\end{array}
$$

- Only two constructors: **A** and **B**
- All values of any sum type built from these constructors
- So $\mathbf{A}(e)$ can have any sum type allowed by e's type
- No need to declare sum types in advance
- Like functions, will "guess the type" in our rules

Sums operational semantics

$$\frac{}{\textbf{match A}(v) \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 \rightarrow e_1[v/x]}$$

$$\frac{}{\textbf{match B}(v) \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 \rightarrow e_2[v/y]}$$

$$\frac{e \rightarrow e'}{\textbf{A}(e) \rightarrow \textbf{A}(e')} \qquad \frac{e \rightarrow e'}{\textbf{B}(e) \rightarrow \textbf{B}(e')}$$

$$\frac{e \rightarrow e'}{\textbf{match } e \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 \rightarrow \textbf{match } e' \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2}$$

match has binding occurrences, just like pattern-matching

(Definition of substitution must avoid capture, just like functions)

What is going on

Feel free to think about *tagged values* in your head:

- A tagged value is a pair of:
  - A tag **A** or **B** (or 0 or 1 if you prefer)
  - The (underlying) value
- A match:
  - Checks the tag
  - Binds the variable to the (underlying) value

This much is just like OCaml and related to homework 2

Sums Typing Rules

Inference version (not trivial to infer; can require annotations)

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \mathsf{A}(e) : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \mathsf{B}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 + \tau_2 \qquad \Gamma, x{:}\tau_1 \vdash e_1 : \tau \qquad \Gamma, y{:}\tau_2 \vdash e_2 : \tau}{\Gamma \vdash \textbf{match } e \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 : \tau}$$

Key ideas:

- For constructor-uses, "other side can be anything"
- For **match**, both sides need same type
  - Don't know which branch will be taken, just like an **if**.
  - In fact, can drop explicit booleans and encode with sums: E.g., **bool** $=$ **int** $+$ **int**, **true** $=$ **A**(0), **false** $=$ **B**(0)

Sums Type Safety

Canonical Forms: If $\cdot \vdash v : \tau_1 + \tau_2$, then there exists a v_1 such that either v is $\mathsf{A}(v_1)$ and $\cdot \vdash v_1 : \tau_1$ or v is $\mathsf{B}(v_1)$ and $\cdot \vdash v_1 : \tau_2$

- Progress for **match** v **with A**$x.\ e_1 \mid$ **B**$y.\ e_2$ follows, as usual, from Canonical Forms
- Preservation for **match** v **with A**$x.\ e_1 \mid$ **B**$y.\ e_2$ follows from the type of the underlying value and the Substitution Lemma
- The Substitution Lemma has new "hard" cases because we have new binding occurrences
- But that's all there is to it (plus lots of induction)

What are sums for?

- Pairs, structs, records, aggregates are fundamental data-builders
- Sums are just as fundamental: “this or that not both”
- You have seen how OCaml does sums (datatypes)
- Worth showing how C and Java do the same thing
  - A primitive in one language is an idiom in another

Sums in C

```
type t = A of t1 | B of t2 | C of t3
match e with A x -> ...
```

One way in C:

```
struct t {
  enum  {A, B, C}            tag;
  union {t1 a; t2 b; t3 c;} data;
};
... switch(e->tag){ case A: t1 x=e->data.a; ...
```

- No static checking that tag is obeyed
- As fat as the fattest variant (avoidable with casts)
  - Mutation costs us again!

Sums in Java

```
type t = A of t1 | B of t2 | C of t3
match e with A x -> ...
```

One way in Java (`t4` is the match-expression's type):

```
abstract class t {abstract t4 m();}
class A extends t { t1 x; t4 m(){...}}
class B extends t { t2 x; t4 m(){...}}
class C extends t { t3 x; t4 m(){...}}
... e.m() ...
```

- A new method in `t` and subclasses for each match expression
- Supports extensibility via new variants (subclasses) instead of extensibility via new operations (**match** expressions)

Pairs vs. Sums

You need both in your language

- With only pairs, you clumsily use dummy values, waste space, and rely on unchecked tagging conventions
- Example: replace $\textbf{int} + (\textbf{int} \rightarrow \textbf{int})$ with $\textbf{int} * (\textbf{int} * (\textbf{int} \rightarrow \textbf{int}))$

Pairs and sums are "logical duals" (more on that later)

- To make a $\tau_1 * \tau_2$ you need a τ_1 *and* a τ_2
- To make a $\tau_1 + \tau_2$ you need a τ_1 *or* a τ_2
- Given a $\tau_1 * \tau_2$, you can get a τ_1 or a τ_2 (or both; your "choice")
- Given a $\tau_1 + \tau_2$, you must be prepared for either a τ_1 or τ_2 (the value's "choice")

Base Types and Primitives, in general

What about floats, strings, ...?
Could add them all or do something more general...

Parameterize our language/semantics by a collection of *base types* $(b_1, \ldots, b_n)$ and *primitives* $(p_1 : \tau_1, \ldots, p_n : \tau_n)$. Examples:

- concat : string$\rightarrow$string$\rightarrow$string
- toInt : float$\rightarrow$int
- "hello" : string

For each primitive, *assume* if applied to values of the right types it produces a value of the right type

Together the types and assumed steps tell us how to type-check and evaluate $p_i\ v_1 \ldots v_n$ where p_i is a primitive

We can prove soundness *once and for all* given the assumptions

Recursion

We won't prove it, but every extension so far preserves termination

A Turing-complete language needs some sort of loop, but our lambda-calculus encoding won't type-check, nor will any encoding of equal expressive power

- So instead add an explicit construct for recursion
- You might be thinking **let rec** $f\ x\ = e$, but we will do something more concise and general but less intuitive

Recursion

We won't prove it, but every extension so far preserves termination

A Turing-complete language needs some sort of loop, but our lambda-calculus encoding won't type-check, nor will any encoding of equal expressive power

- So instead add an explicit construct for recursion
- You might be thinking **let rec** $f\ x\ = e$, but we will do something more concise and general but less intuitive

$$e ::= \ldots \mid \textbf{fix}\ e$$

$$\frac{e \to e'}{\textbf{fix}\ e \to \textbf{fix}\ e'} \qquad \frac{}{\textbf{fix}\ \lambda x.\ e \to e[\textbf{fix}\ \lambda x.\ e/x]}$$

No new values and no new types

Using fix

To use **fix** like **let rec**, just pass it a two-argument function where the first argument is for recursion

- Not shown: **fix** and tuples can also encode mutual recursion

Example:
$\mathbf{(}\textbf{fix}\ \boldsymbol{\lambda f.\ \lambda n.}\ \textbf{if}\ \boldsymbol{(n{<}1)\ 1\ (n * (f(n-1))))\ 5}$

Using fix

To use **fix** like **let rec**, just pass it a two-argument function where the first argument is for recursion

- Not shown: **fix** and tuples can also encode mutual recursion

Example:

$$(\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n<1)\ 1\ (n*(f(n-1))))\ 5$$

$$\rightarrow$$

$$(\lambda n.\ \textbf{if}\ (n<1)\ 1\ (n*((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n<1)\ 1\ (n*(f(n-1))))(n-1))))\ 5$$

Using fix

To use **fix** like **let rec**, just pass it a two-argument function where the first argument is for recursion

- Not shown: **fix** and tuples can also encode mutual recursion

Example:

$$(\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))\ 5$$

$$\rightarrow$$

$$(\lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(n-1))))\ 5$$

$$\rightarrow$$

$$\textbf{if}\ (5{<}1)\ 1\ (5 * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(5-1))$$

Using fix

To use **fix** like **let rec**, just pass it a two-argument function where the first argument is for recursion

- Not shown: **fix** and tuples can also encode mutual recursion

Example:

$$(\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))\ 5$$

$$\rightarrow$$

$$(\lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(n-1))))\ 5$$

$$\rightarrow$$

$$\textbf{if}\ (5{<}1)\ 1\ (5 * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(5-1))$$

$$\rightarrow^2$$

$$5 * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(5-1))$$

Using fix

To use **fix** like **let rec**, just pass it a two-argument function where the first argument is for recursion

- Not shown: **fix** and tuples can also encode mutual recursion

Example:

$(\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))\ 5$

$\rightarrow$

$(\lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(n-1))))\ 5$

$\rightarrow$

$\textbf{if}\ (5{<}1)\ 1\ (5 * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(5-1)))$

$\rightarrow^2$

$5 * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(5-1))$

$\rightarrow^2$

$5 * ((\lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * ((\textbf{fix}\ \lambda f.\ \lambda n.\ \textbf{if}\ (n{<}1)\ 1\ (n * (f(n-1))))(n-1))))\ 4)$

$\rightarrow$

...

Why called fix?

In math, a fix-point of a function g is an x such that $g(x) = x$

- This makes sense only if g has type $\tau \to \tau$ for some τ
- A particular g could have have 0, 1, 39, or infinity fix-points
- Examples for functions of type **int** $\to$ **int**:
  - $\lambda x.\ x + 1$ has no fix-points
  - $\lambda x.\ x * 0$ has one fix-point
  - $\lambda x.$ absolute_value(x) has an infinite number of fix-points
  - $\lambda x.$ **if** $(x < 10\ \&\&\ x > 0)\ x\ 0$ has 10 fix-points

Higher types

At higher types like $(\textbf{int} \rightarrow \textbf{int}) \rightarrow (\textbf{int} \rightarrow \textbf{int})$, the notion of fix-point is exactly the same (but harder to think about)

- For what inputs f of type $\textbf{int} \rightarrow \textbf{int}$ is $g(f) = f$

Examples:

- $\lambda f.\ \lambda x.\ (f\ x) + 1$ has no fix-points
- $\lambda f.\ \lambda x.\ (f\ x) * 0$ (or just $\lambda f.\ \lambda x.\ 0$) has 1 fix-point
  - The function that always returns 0
  - In math, there is exactly one such function (cf. equivalence)
- $\lambda f.\ \lambda x.$ absolute_value$(f\ x)$ has an infinite number of fix-points: Any function that never returns a negative result

Back to factorial

Now, what are the fix-points of
$\lambda f.\ \lambda x.\ \textbf{if}\ (x < 1)\ 1\ (x * (f(x - 1)))$?

It turns out there is exactly one (in math): the factorial function!

And **fix** $\lambda f.\ \lambda x.\ \textbf{if}\ (x < 1)\ 1\ (x * (f(x - 1)))$ behaves just like the factorial function

- That is, it behaves just like the fix-point of
  $\lambda f.\ \lambda x.\ \textbf{if}\ (x < 1)\ 1\ (x * (f(x - 1)))$
- In general, **fix** takes a function-taking-function and returns its fix-point

(This isn't necessarily important, but it explains the terminology and shows that programming is deeply connected to mathematics)

Typing fix

$$\frac{\Gamma \vdash e : \tau \to \tau}{\Gamma \vdash \textbf{fix } e : \tau}$$

Math explanation: If e is a function from τ to τ, then **fix** e, the fixed-point of e, is some τ with the fixed-point property

- So it's something with type τ

Operational explanation: **fix** $\lambda x.\ e'$ becomes e'[**fix** $\lambda x.\ e'/x$]

- The substitution means x and **fix** $\lambda x.\ e'$ need the same type
- The result means e' and **fix** $\lambda x.\ e'$ need the same type

Note: The τ in the typing rule is usually insantiated with a function type

- e.g., $\tau_1 \to \tau_2$, so e has type $(\tau_1 \to \tau_2) \to (\tau_1 \to \tau_2)$

Note: Proving soundness is straightforward!

General approach

We added let, booleans, pairs, records, sums, and fix

- **let** was syntactic sugar
- **fix** made us Turing-complete by “baking in” self-application
- The others *added types*

Whenever we add a new form of type $\boldsymbol{\tau}$ there are:

- Introduction forms (ways to make values of type $\boldsymbol{\tau}$)
- Elimination forms (ways to use values of type $\boldsymbol{\tau}$)

What are these forms for functions? Pairs? Sums?

When you add a new type, think “what are the intro and elim forms”?

Anonymity

We added many forms of types, all *unnamed* a.k.a. *structural*.
Many real PLs have (all or mostly) *named* types:

- Java, C, C++: all record types (or similar) have names
  - Omitting them just means compiler makes up a name
- OCaml sum types and record types have names

A never-ending debate:

- Structual types allow more code reuse: good
- Named types allow less code reuse: good
- Structural types allow generic type-based code: good
- Named types let type-based code distinguish names: good

The theory is often easier and simpler with structural types

Termination

Surprising fact: If $\cdot \vdash e : \tau$ in STLC with all our additions *except* **fix**, then there exists a v such that $e \rightarrow^* v$

- That is, all programs terminate

So termination is trivially decidable (the constant "yes" function), so our language is not Turing-complete

The proof requires more advanced techniques than we have learned so far because the size of expressions and typing derivations does not decrease with each program step

- Could present it in about an hour if desired

Non-proof:

- Recursion in λ calculus requires some sort of self-application
- Easy fact: For all Γ, x, and τ, we *cannot* derive $\Gamma \vdash x\ x : \tau$