Types and Type Safety

Types as a way to avoid errors

- We have discussed this before

- Expressions like $1 + \texttt{true}$ with unspecified behavior are rejected by the type system and are not allowed to execute

- Not every runtime error can be rejected by the type system. Example: $1/0$

Type checking to avoid errors

- If the goal is to eliminate errors then anything that is guaranteed not to cause an error should be accepted by the type system

- This violates the whole concept of abstract data types.

Stack ADT

```
module Stack (Stack, empty, push, pop) where

-- Internal representation type; not exported
data Stack a = Stack [a]

empty = Stack []

push a (Stack as) = Stack (a : as)

pop (Stack []) = error "Empty stack"
pop (Stack (a:as)) = (a, Stack as)
```

Using the ADT

```
module C where

import Stack

s1 :: Stack Int
s1 = push 1 (push 2 (push 3 empty))

bad :: Int
bad = let Stack elements = s1 in length elements
```

Types as a programming discipline

- Even if the type system has information that the code is safe; it might still refuse to propagate the information and reject the code

- Modern type systems are all about controlling access to information as oppposed to avoiding run-time errors.

- This makes type systems relevant to encapsulation, security, proof-carrying code, etc

Type Soundness

A little functional language

- Syntax of expressions

$$
\begin{aligned}
e &::= c \mid x \mid \lambda x.e \mid e_1 e_2 \\
c &::= 0 \mid 1 \mid \cdots \mid \textsf{add1} \mid \textsf{true} \mid \textsf{false} \mid \textsf{not}
\end{aligned}
$$

- Syntax of types

$$\tau ::= \textsf{int} \mid \textsf{bool} \mid \cdots \mid \tau \rightarrow \tau$$

Semantics

- Syntax of values

$$v ::= c \mid x \mid \lambda x.e$$

- Small-step reductions:

$$\begin{aligned}
\textsf{add1}\ 0 &\longrightarrow 1 \\
\textsf{add1}\ 1 &\longrightarrow 2 \\
&\vdots \\
\textsf{not false} &\longrightarrow \textsf{true} \\
\textsf{not true} &\longrightarrow \textsf{false} \\
\\
(\lambda x.e)v &\longrightarrow e[v/x]
\end{aligned}$$

Examples

$$
\begin{aligned}
(\lambda x.\mathsf{add1}\ x)\ 3 &\longrightarrow \mathsf{add1}\ 3 \\
&\longrightarrow 4
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.(\lambda y.\mathsf{add1}\ y)\ x)\ 3 &\longrightarrow (\lambda y.\mathsf{add1}\ y)\ 3 \\
&\longrightarrow \mathsf{add1}\ 3 \\
&\longrightarrow 4
\end{aligned}
$$

$$
\begin{aligned}
(\lambda x.\mathsf{not}\ x)\ \mathsf{true} &\longrightarrow \mathsf{not\ true} \\
&\longrightarrow \mathsf{false}
\end{aligned}
$$

Examples

- $((\lambda x.\lambda y.\mathsf{add1}\ x)\ (\mathsf{add1}\ 3))\ (\mathsf{add1}\ 6) \quad \longrightarrow \quad ???$

- Technically this is not related by $\longrightarrow$ to anything

- Need to explain how to search for a place where to apply a reduction

Evaluation contexts

- Syntax of evaluation contexts

$$E ::= [] \mid Ee \mid vE$$

- Small-step evaluation

$$E[e] \mapsto E[e'] \qquad \text{iff} \qquad e \rightarrow e'$$

- Big-step evaluation

$$eval(e) = v \qquad \text{iff} \qquad e \mapsto^* v$$

Decomposition

- $((\lambda x.\lambda y.\textsf{add1}\ x)\ (\textsf{add1}\ 3))\ (\textsf{add1}\ 6)$

- Can be decomposed into: $E[r]$ where:
  - $E = [\,]$, $r = ((\lambda x.\lambda y.\textsf{add1}\ x)\ (\textsf{add1}\ 3))\ (\textsf{add1}\ 6)$
  - $E = ([\,]\ (\textsf{add1}\ 6))$, $r = (\lambda x.\lambda y.\textsf{add1}\ x)\ (\textsf{add1}\ 3)$
  - $E = (([\,]\ (\textsf{add1}\ 3))\ (\textsf{add1}\ 6))$, $r = \lambda x.\lambda y.\textsf{add1}\ x$
  - $E = ((\lambda x.\lambda y.\textsf{add1}\ x)\ [\,])\ (\textsf{add1}\ 6)$, $r = (\textsf{add1}\ 3)$

Unique decomposition lemma

- Every e can be uniquely represented as either:
  - a value v
  - a decomposition $E[r]$ for some evaluation context E and expression r where:
    - r is variable x, or
    - r is a redex, or
    - r is a faulty expression
- Redexes: $\mathsf{add1}\ n \mid \mathsf{not}\ b \mid (\lambda x.e)v$
- Faulty:

$$\begin{array}{l} n\ e \mid \mathsf{true}\ e \mid \mathsf{false}\ e \mid \\ \mathsf{add1}\ b \mid \mathsf{add1}\ (\lambda x.e) \mid \\ \mathsf{not}\ n \mid \mathsf{not}\ (\lambda x.e) \end{array}$$

Examples

- $((\lambda x.\lambda y.\mathsf{add1}\ x)\ (\mathsf{add1}\ 3))\ (\mathsf{add1}\ 6)$
- Can be decomposed into: $E[r]$ where:
  - $E = []$, $r = ((\lambda x.\lambda y.\mathsf{add1}\ x)\ (\mathsf{add1}\ 3))\ (\mathsf{add1}\ 6)$
    No good
  - $E = ([]\ (\mathsf{add1}\ 6))$, $r = (\lambda x.\lambda y.\mathsf{add1}\ x)\ (\mathsf{add1}\ 3)$
    No good
  - $E = (([]\ (\mathsf{add1}\ 3))\ (\mathsf{add1}\ 6))$, $r = \lambda x.\lambda y.\mathsf{add1}\ x$
    No good
  - $E = ((\lambda x.\lambda y.\mathsf{add1}\ x)\ [])\ (\mathsf{add1}\ 6)$, $r = (\mathsf{add1}\ 3)$
    $E[r]$ where r is a redex. Good

Examples

- 5 decomposes as the value 5
- x decomposes as $\boxed{x}$
- add1 (not 5) decomposes as add1 $\boxed{\text{(not 5)}}$ which focuses on a faulty expression
- add1 (add1 5) decomposes as add1 $\boxed{\text{(add1 5)}}$ which focuses on a redex
- $(\lambda y.2)\ (\lambda x.\text{add1 true})$ decomposes as $\boxed{(\lambda y.2)\ (\lambda x.\text{add1 true})}$. (It contains a faulty expression though!)

Examples

$$
\begin{array}{ll}
 & ((\lambda x.\lambda y.\textsf{add1}\ x)\ (\textsf{add1}\ 3))\ (\textsf{add1}\ 6) \\
\equiv & ((\lambda x.\lambda y.\textsf{add1}\ x)\ \boxed{(\textsf{add1}\ 3)})\ (\textsf{add1}\ 6) \\
\longmapsto & ((\lambda x.\lambda y.\textsf{add1}\ x)\ 4)\ (\textsf{add1}\ 6) \\
\equiv & \boxed{((\lambda x.\lambda y.\textsf{add1}\ x)\ 4)}\ (\textsf{add1}\ 6) \\
\longmapsto & (\lambda y.\textsf{add1}\ 4)\ (\textsf{add1}\ 6) \\
\equiv & (\lambda y.\textsf{add1}\ 4)\ \boxed{(\textsf{add1}\ 6)} \\
\longmapsto & (\lambda y.\textsf{add1}\ 4)\ 7 \\
\equiv & \boxed{(\lambda y.\textsf{add1}\ 4)\ 7} \\
\longmapsto & \textsf{add1}\ 4 \\
\equiv & \boxed{\textsf{add1}\ 4} \\
\longmapsto & 5
\end{array}
$$

Big picture

- At every step, if we have not reached a final answer, we attempt a decomposition of the current term
- The lemma says the decomposition can give us focus on a free variable, a redex, or a faulty expression
- If the source program is closed, we should never encounter free variables
- If the source program is well-typed, we should never encounter faulty expressions
- If the source program is closed and well-typed, we can always make progress until we reach a value (or diverge)

Relate typing to evaluation

- Must define what it means for an expression to typecheck

- Must show that typing is preserved by reduction

- Must also show that faulty expressions are not typable

Type system

$$\frac{}{\Gamma, x{:}\,\tau \vdash x{:}\,\tau}\ Axv \qquad \frac{}{\Gamma \vdash n{:}\,\mathsf{int}}\ Axi$$

$$\frac{}{\Gamma \vdash \mathsf{true}{:}\,\mathsf{bool}}\ Axt \qquad \frac{}{\Gamma \vdash \mathsf{false}{:}\,\mathsf{bool}}\ Axf$$

$$\frac{}{\Gamma \vdash \mathsf{add1}{:}\,\mathsf{int}{\longrightarrow}\mathsf{int}}\ Axa \qquad \frac{}{\Gamma \vdash \mathsf{not}{:}\,\mathsf{bool}{\longrightarrow}\mathsf{bool}}\ Axn$$

$$\frac{\Gamma, x{:}\,\tau \vdash e{:}\,\tau'}{\Gamma \vdash \lambda x.e{:}\,\tau{\longrightarrow}\tau'}\ {\longrightarrow_i}$$

$$\frac{\Gamma \vdash e_1{:}\,\tau'{\longrightarrow}\tau \quad \Gamma \vdash e_2{:}\,\tau'}{\Gamma \vdash e_1 e_2{:}\,\tau}\ {\longrightarrow_e}$$

Typing detour

- Can we typecheck $(\lambda x.x\ x)\ (\lambda x.x\ x)$?

- If it does typecheck then every subterm must also typecheck!

- Let's try to typecheck $x\ x$:

$$\frac{\Gamma \vdash x\colon \tau' \to \tau \quad \Gamma \vdash x\colon \tau'}{\Gamma \vdash x\ x\colon \tau}$$

- So we must have $\tau' \to \tau = \tau'$ which is impossible in our system

Typing recursion

- Many approaches: Add recursive type definitions, or polymorphism, or references, etc

- Add a construct fix with the following reduction rule:

$$\mathsf{fix}\ f.e \rightarrow e[(\lambda x.(\mathsf{fix}\ f.e)\ x)/f]$$

- Intuition. The reduction is almost:

$$\mathsf{fix}\ f.e \rightarrow e[\mathsf{fix}\ f.e/f]$$

- The λ delays the unfolding of the recursion until it is needed

Example

Let $\mathit{fact} = (\textsf{fix}\ f.\lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (f\ (n-1)))$:

$$
\begin{array}{ll}
 & \mathit{fact}\ 2 \\
\longrightarrow & (\lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * ((\lambda x.\mathit{fact}\ x)\ (n-1)))\ 2 \\
\longrightarrow & \texttt{if}\ 2 = 0\ \texttt{then}\ 1\ \texttt{else}\ 2 * ((\lambda x.\mathit{fact}\ x)\ (2-1)) \\
\longrightarrow & 2 * ((\lambda x.\mathit{fact}\ x)\ 1) \\
\longrightarrow & 2 * (\mathit{fact}\ 1) \\
\longrightarrow & 2 * ((\lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * ((\lambda x.\mathit{fact}\ x)\ (n-1)))\ 1) \\
\longrightarrow & 2 * (\texttt{if}\ 1 = 0\ \texttt{then}\ 1\ \texttt{else}\ 1 * ((\lambda x.\mathit{fact}\ x)\ (1-1))) \\
\longrightarrow & 2 * (1 * ((\lambda x.\mathit{fact}\ x)\ 0)) \\
\longrightarrow & 2 * (1 * (\mathit{fact}\ 0)) \\
\longrightarrow & 2 * (1 * ((\lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * ((\lambda x.\mathit{fact}\ x)\ (n-1)))\ 0)) \\
\longrightarrow & 2 * (1 * (\texttt{if}\ 0 = 0\ \texttt{then}\ 1\ \texttt{else}\ 0 * ((\lambda x.\mathit{fact}\ x)\ (0-1)))) \\
\longrightarrow & 2 * (1 * 1)
\end{array}
$$

Type rule for fix

- Look at the reduction

$$\mathsf{fix}\ f.e \longrightarrow e[(\lambda x.(\mathsf{fix}\ f.e)\ x)/f]$$

- In the RHS, $\mathsf{fix}\ f.e$ is applied so it must have type: $\tau{\rightarrow}\tau'$ for some τ and τ'; we must also have $x{:}\,\tau$.
- In the RHS, f is substituted by something of type $\tau{\rightarrow}\tau'$
- The variable f may occur free in e with type $\tau{\rightarrow}\tau'$
- If evaluation is to preserve types, the types of the LHS and RHS must be the same:

$$\frac{\Gamma, f{:}\,\tau{\rightarrow}\tau' \vdash e{:}\,\tau{\rightarrow}\tau'}{\Gamma \vdash \mathsf{fix}\ f.e{:}\,\tau{\rightarrow}\tau'}$$

Example

- The general rule:

$$\frac{\Gamma, f\colon \tau \to \tau' \vdash e\colon \tau \to \tau'}{\Gamma \vdash \textsf{fix}\ f.e\colon \tau \to \tau'}$$

- $fact = (\textsf{fix}\ f.\lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (f\ (n-1)))$

$$\cfrac{\cfrac{\ldots \quad \cfrac{\vdots}{f\colon \textsf{int}\to\textsf{int}, n\colon \textsf{int} \vdash n * (f\ (n-1))\colon \textsf{int}}}{f\colon \textsf{int}\to\textsf{int}, n\colon \textsf{int} \vdash \texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (f\ (n-1))\colon \textsf{int}}}{\cfrac{f\colon \textsf{int}\to\textsf{int} \vdash \lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (f\ (n-1))\colon \textsf{int}\to\textsf{int}}{\vdash (\textsf{fix}\ f.\lambda n.\texttt{if}\ n = 0\ \texttt{then}\ 1\ \texttt{else}\ n * (f\ (n-1)))\colon \textsf{int}\to\textsf{int}}}$$

Infinite loop

- Can we type an infinite loop now?
- Consider $\textsf{fix}\ x.x$

$$\frac{\Gamma, x\colon \tau{\longrightarrow}\tau' \vdash x\colon \tau{\longrightarrow}\tau'}{\Gamma \vdash \textsf{fix}\ x.x\colon \tau{\longrightarrow}\tau'}$$

- Running $(\textsf{fix}\ x.x)\ 0$:

$$\begin{array}{ll} & (\textsf{fix}\ x.x)\ 0 \\ \longrightarrow & (x[(\lambda y.(\textsf{fix}\ x.x)\ y)/x])\ 0 \\ \equiv & (\lambda y.(\textsf{fix}\ x.x)\ y)\ 0 \\ \longrightarrow & (\textsf{fix}\ x.x)\ 0 \\ \longrightarrow & \ldots \end{array}$$

- The type of the infinite loop is τ' for any type you want!
- You can use it in any context you want $1 + [\,]$, $\textsf{not}[\,]$, etc

A little functional language (revisited)

- Syntax of expressions

$$
\begin{aligned}
e &::= c \mid x \mid \lambda x.e \mid e_1 e_2 \mid \textsf{fix}\ f.e \\
c &::= 0 \mid 1 \mid \cdots \mid \textsf{add1} \mid \textsf{true} \mid \textsf{false} \mid \textsf{not}
\end{aligned}
$$

- Syntax of types

$$
\tau ::= \textsf{int} \mid \textsf{bool} \mid \cdots \mid \tau \rightarrow \tau
$$

Type safety

- The goal is to prove:

  If $\vdash e\!:\tau$ then either the evaluation of e diverges or $eval(e) = v$ and $\vdash v\!:\tau$

- Two fundamental lemmas needed
- Progress: (Well-typed terms do not get stuck or do not encounter configurations whose behavior is not specified by the semantics)

  If $\vdash e\!:\tau$ then either e is a value or there exists an e' such that $e \mapsto e'$

- Subject Reduction: (Every step of evaluation preserves the above property)

  If $\vdash e\!:\tau$ and $e \mapsto e'$ then $\vdash e'\!:\tau$

Subject reduction

- The heart of the proof really

- We look at each reduction in turn; assume the LHS typechecks; use that knowledge to show that some type derivations must exist and use them to construct a type derivation for the RHS

- Very similar to the proofs we have done for operational semantics (Assignment 2).

Example

- Consider a simple instance of our evaluation rules:

$$(\lambda x.e)\ (\textsf{add1}\ 3) \longrightarrow (\lambda x.e)\ 4$$

- Assume the LHS typechecks:

$$\frac{\Gamma \vdash (\lambda x.e) \colon \textsf{int} \!\longrightarrow\! \tau \quad \Gamma \vdash (\textsf{add1}\ 3) \colon \textsf{int}}{\Gamma \vdash (\lambda x.e)\ (\textsf{add1}\ 3) \colon \tau}$$

  We know that we have a derivation of $\Gamma \vdash (\lambda x.e) \colon \textsf{int} \!\longrightarrow\! \tau$

- Use the derivation above, we can construct a derivation for the RHS:

$$\frac{\Gamma \vdash (\lambda x.e) \colon \textsf{int} \!\longrightarrow\! \tau \quad \Gamma \vdash 4 \colon \textsf{int}}{\Gamma \vdash (\lambda x.e)\ 4 \colon \tau}$$

Proof of subject reduction

- Recall the definition of evaluation contexts:

$$E ::= [] \mid Ee \mid vE$$

- Our reductions are:

$$\begin{array}{rcl}
E[\textsf{add1}\ 0] & \mapsto & E[1] \\
E[\textsf{add1}\ 1] & \mapsto & E[2] \\
& \vdots & \\
E[\textsf{not false}] & \mapsto & E[\textsf{true}] \\
E[\textsf{not true}] & \mapsto & E[\textsf{false}] \\
\\
E[(\lambda x.e)v] & \mapsto & E[e[v/x]] \\
\\
E[\textsf{fix}\ f.e] & \mapsto & E[e[(\lambda x.(\textsf{fix}\ f.e)\ x)/f]]
\end{array}$$

Case I

- Assume $\Gamma \vdash E[\texttt{add1}\ 0] : \tau$. Show that $\Gamma \vdash E[1] : \tau$

- What do we know about the type derivation $\Gamma \vdash E[\texttt{add1}\ 0] : \tau$

- Not much unless we look at cases for E

- Definition of E is recursive; proof goes by induction on the definition of E

Case I (continued)

- Prove by induction on E that:

  If $\Gamma \vdash E[\texttt{add1}\ 0] : \tau$ then $\Gamma \vdash E[1] : \tau$

- $E = []$. We have $\Gamma \vdash \texttt{add1}\ 0 : \tau$ and we want to show $\Gamma \vdash 1 : \tau$.

- The only way we could possibly derive $\Gamma \vdash \texttt{add1}\ 0 : \tau$ is for $\tau = \texttt{int}$. In that case, we can clearly derive a typing for the RHS.

Case I (continued)

- Still proving by induction on E that:

  If $\Gamma \vdash E[\textsf{add1}\ 0]\colon \tau$ then $\Gamma \vdash E[1]\colon \tau$

- $E = E'e$. We have $\Gamma \vdash E'[\textsf{add1}\ 0]\ e\colon \tau$. Show that $\Gamma \vdash E'[1]\ e\colon \tau$.
- The derivation we are given must look like:

$$\frac{\Gamma \vdash E'[\textsf{add1}\ 0]\colon \tau' \rightarrow \tau \quad \Gamma \vdash e\colon \tau'}{\Gamma \vdash E'[\textsf{add1}\ 0]\ e\colon \tau}$$

- We want to build a derivation:

$$\frac{\Gamma \vdash E'[1]\colon \tau' \rightarrow \tau \quad \Gamma \vdash e\colon \tau'}{\Gamma \vdash E'[1]\ e\colon \tau}$$

- Induction hypothesis tells us that $\Gamma \vdash E'[1]\colon \tau' \rightarrow \tau$ because $\Gamma \vdash E'[\textsf{add1}\ 0]\colon \tau' \rightarrow \tau$.

Other cases

- We can finish the proof of this case easily
- The proof of this case $E[\textsf{not false}] \mapsto E[\textsf{true}]$ would be almost identical.
- Should have generalized the previous proof.
- What's the general statement that would allow us to conclude:

$$\begin{array}{l}
\text{If } \Gamma \vdash E[\textsf{add1 } 0] : \tau \text{ then } \Gamma \vdash E[1] : \tau \\
\text{If } \Gamma \vdash E[\textsf{not false}] : \tau \text{ then } \Gamma \vdash E[\textsf{true}] : \tau \\
\text{If } \Gamma \vdash E[(\lambda x.e)v] : \tau \text{ then } \Gamma \vdash E[e[v/x]] : \tau \\
\text{If } \Gamma \vdash E[\textsf{fix } f.e] : \tau \text{ then } \Gamma \vdash E[e[(\lambda x.(\textsf{fix } f.e)\ x)/f]] : \tau
\end{array}$$

Replacement lemma

- In general we are given $\Gamma \vdash E[e] : \tau$
- This implies that e must typecheck but not necessarily in the same environment
- In general all we know is that for some Γ' and τ' we can prove $\Gamma' \vdash e : \tau'$
- In general we want to replace e by some expression e' of the same type. In other words an expression e' such that we can independently prove that $\Gamma' \vdash e' : \tau'$
- We want a general lemma that says that this is always ok.

Replacement lemma

$$\dfrac{\Gamma' \vdash e : \tau' \quad \ldots \quad \ldots \quad \Gamma' \vdash e : \tau'}{\Gamma \vdash E[e] : \tau}$$

$$\dfrac{\Gamma' \vdash e' : \tau' \quad \ldots \quad \ldots \quad \Gamma' \vdash e' : \tau'}{\Gamma \vdash E[e'] : \tau}$$

Seems obvious

- What's the big deal?
- Consider a language with exceptions: for example a language where division by zero throws an exception: err

$$\begin{array}{rcl} \mathsf{div}\ 6\ 2 & \longrightarrow & 3 \\ & \vdots & \\ \mathsf{div}\ n\ 0 & \longrightarrow & \mathsf{err} \\ & \vdots & \end{array}$$

- Assume that x and y are of type int. It is reasonable to assume that all of the following expressions typecheck:

$$\begin{array}{l} \mathsf{add1}\ (\mathsf{div}\ x\ y) \\ \mathsf{not}\ (\mathsf{div}\ x\ y = 3) \\ (\texttt{if}\ (\mathsf{div}\ x\ y = 3)\ \texttt{then}\ (\lambda a.a)\ \texttt{else}\ (\lambda a.a))\ 5 \end{array}$$

Propagating exceptions

- Consider what happens at runtime if $x = 1$ and $y = 0$.

- The first expression evaluates as follows:

$$\mathsf{add1}\ (\mathsf{div}\ 1\ 0) \ \mapsto\ \mathsf{add1}\ \mathsf{err}$$

- The second expression evaluates as follows:

$$\begin{aligned} \mathsf{not}\ (\mathsf{div}\ 1\ 0 = 3) &\ \mapsto\ \mathsf{not}\ (\mathsf{err} = 3) \\ &\ \mapsto\ \mathsf{not}\ \mathsf{err} \end{aligned}$$

Propagating exceptions

- The third expression evaluates as follows:

$$\begin{array}{rl} & (\texttt{if}\ (\mathsf{div}\ 1\ 0 = 3)\ \texttt{then}\ (\lambda a.a)\ \texttt{else}\ (\lambda a.a))\ 5 \\ \mapsto & (\texttt{if}\ (\mathsf{err} = 3)\ \texttt{then}\ (\lambda a.a)\ \texttt{else}\ (\lambda a.a))\ 5 \\ \mapsto & (\texttt{if}\ \mathsf{err}\ \texttt{then}\ (\lambda a.a)\ \texttt{else}\ (\lambda a.a))\ 5 \\ \mapsto & \mathsf{err}\ 5 \end{array}$$

- So at runtime, err might appear in a context expecting an int, a bool, or even a function

- For any τ we have:

$$\overline{\Gamma \vdash \mathsf{err} : \tau}$$

Replacement lemma again

- We are given:
  - $\Gamma \vdash E[\mathsf{err}] \colon \mathsf{bool}$
  - $\Gamma' \vdash \mathsf{err} \colon \mathsf{int}$
  - $\Gamma' \vdash 5 \colon \mathsf{int}$
- By the replacement lemma conclude $\Gamma \vdash E[5] \colon \mathsf{bool}$
- Our conclusion is wrong!
- Take $E = [\,]$. The assumptions are:
  - $\Gamma \vdash \mathsf{err} \colon \mathsf{bool}$
  - $\Gamma' \vdash \mathsf{err} \colon \mathsf{int}$
  - $\Gamma' \vdash 5 \colon \mathsf{int}$

  We have concluded: $\Gamma \vdash 5 \colon \mathsf{bool}$ which is bogus.