Lecture 9
How to make a sound type system
Why types are good

Prevent errors: many simple errors caught by types

Ensure memory safety or other desired properties

Document the program (purpose of parameters)

Make it easier to change

Make compilation more efficient: remove checks, specialize
An unsound (broken) type system

A type system that aims to ensure some property but, in fact, fails.

For example: suppose we have a system that aims to ensure that if parameter is of type `Int`, then it is only invoked with values of type `Int`. But we find a (tricky) program that passes the type checker and ends up invoking the function with the reference to a string. This is unsoundness.

Sometimes unsoundness is (somewhat) intentional compromise:

- type casts in C
- covariance for function arguments and arrays

Often unintentional (unsoundness type system bugs) due to subtle interactions between e.g. subtyping, generics, mutation, higher-order functions, recursion
Goal today

Define precisely a small language:

- its abstract syntax (as certain math expressions)
- its operational semantics (interpreter written in math)
- its type rules

Show that our type system prevents certain kinds of errors
Inductively defined relation: example

Define relation \( r \subseteq \mathbb{Z} \times \mathbb{Z} \) using these inductive rules.

\[
\begin{align*}
(0, 0) & \in r \quad \text{(zero)} \\
(x, y) & \in r \\
& \quad \Rightarrow (x, y + 1) \in r \quad \text{(increase right)} \\
(x, y) & \in r \\
& \quad \Rightarrow (x + 1, y + 1) \in r \quad \text{(increase both)} \\
(x, y) & \in r \\
& \quad \Rightarrow (x - 1, y - 1) \in r \quad \text{(decrease both)}
\end{align*}
\]

Which relations satisfy these rules?
Inductively defined relation: example

Define relation $r \subseteq \mathbb{Z} \times \mathbb{Z}$ using these inductive rules.

\[
\begin{align*}
(0, 0) &\in r \quad \text{(zero)} \\
(x, y) &\in r \quad \Rightarrow \quad (x, y + 1) \in r \quad \text{(increase right)} \\
(x, y) &\in r \quad \Rightarrow \quad (x + 1, y + 1) \in r \quad \text{(increase both)} \\
(x, y) &\in r \quad \Rightarrow \quad (x - 1, y - 1) \in r \quad \text{(decrease both)}
\end{align*}
\]

Which relations satisfy these rules?

\[
\begin{align*}
\Rightarrow r = \{(x, y) \mid x = 0 \lor y = 0\} \quad ?
\end{align*}
\]
Inductively defined relation: example

Define relation \( r \subseteq \mathbb{Z} \times \mathbb{Z} \) using these inductive rules.

\[
\begin{align*}
(0, 0) &\in r \quad \text{(zero)} \\
(x, y) &\in r \quad \Rightarrow \quad (x, y + 1) \in r \quad \text{(increase right)} \\
(x, y) &\in r \quad \Rightarrow \quad (x + 1, y + 1) \in r \quad \text{(increase both)} \\
(x, y) &\in r \quad \Rightarrow \quad (x - 1, y - 1) \in r \quad \text{(decrease both)}
\end{align*}
\]

Which relations satisfy these rules?

\[ r = \{(x, y) \mid x = 0 \lor y = 0\} \? \text{ No} \]
Inductively defined relation: example

Define relation $r \subseteq \mathbb{Z} \times \mathbb{Z}$ using these inductive rules.

- $(0, 0) \in r$ (zero)
- $(x, y) \in r \implies (x, y + 1) \in r$ (increase right)
- $(x, y) \in r \implies (x + 1, y + 1) \in r$ (increase both)
- $(x, y) \in r \implies (x - 1, y - 1) \in r$ (decrease both)

Which relations satisfy these rules?

- $r = \{(x, y) \mid x = 0 \lor y = 0\}$ ? No
- $r = \{(x, y) \mid x \leq 0 \land 0 \leq y\}$ ?
Inductively defined relation: example

Define relation \( r \subseteq \mathbb{Z} \times \mathbb{Z} \) using these inductive rules.

\[
(0, 0) \in r \quad \text{(zero)}
\]

\[
(x, y) \in r \quad \Rightarrow \quad (x, y + 1) \in r \quad \text{(increase right)}
\]

\[
(x, y) \in r \quad \Rightarrow \quad (x + 1, y + 1) \in r \quad \text{(increase both)}
\]

\[
(x, y) \in r \quad \Rightarrow \quad (x - 1, y - 1) \in r \quad \text{(decrease both)}
\]

Which relations satisfy these rules?

\[ r = \{(x, y) \mid x = 0 \lor y = 0\} \quad \text{? No} \]

\[ r = \{(x, y) \mid x \leq 0 \land 0 \leq y\} \quad \text{? No} \]
Inductively defined relation: example

Define relation \( r \subseteq \mathbb{Z} \times \mathbb{Z} \) using these inductive rules.

\[
\begin{align*}
(0, 0) & \in r \quad \text{(zero)} \\
(x, y) & \in r \implies (x, y + 1) \in r \quad \text{(increase right)} \\
(x, y) & \in r \implies (x + 1, y + 1) \in r \quad \text{(increase both)} \\
(x, y) & \in r \implies (x - 1, y - 1) \in r \quad \text{(decrease both)}
\end{align*}
\]

Which relations satisfy these rules?

\[ r = \{ (x, y) | x = 0 \lor y = 0 \} \quad ? \quad \text{No} \]
\[ r = \{ (x, y) | x \leq 0 \land 0 \leq y \} \quad ? \quad \text{No} \]
\[ r = \mathbb{Z} \times \mathbb{Z} \quad ? \]
Inductively defined relation: example

Define relation $r \subseteq \mathbb{Z} \times \mathbb{Z}$ using these inductive rules.

\[
(0, 0) \in r \quad \text{(zero)}
\]
\[
(x, y) \in r \quad \Rightarrow \quad (x, y + 1) \in r \quad \text{(increase right)}
\]
\[
(x, y) \in r \quad \Rightarrow \quad (x + 1, y + 1) \in r \quad \text{(increase both)}
\]
\[
(x, y) \in r \quad \Rightarrow \quad (x - 1, y - 1) \in r \quad \text{(decrease both)}
\]

Which relations satisfy these rules?

- $r = \{ (x, y) \mid x = 0 \lor y = 0 \}$ ? No
- $r = \{ (x, y) \mid x \leq 0 \land 0 \leq y \}$ ? No
- $r = \mathbb{Z} \times \mathbb{Z}$ ? Yes
Inductively defined relation: example

Define relation $r \subseteq \mathbb{Z} \times \mathbb{Z}$ using these inductive rules.

\[
(0, 0) \in r \quad \text{(zero)}
\]

\[
(x, y) \in r \implies (x, y + 1) \in r \quad \text{(increase right)}
\]

\[
(x, y) \in r \implies (x + 1, y + 1) \in r \quad \text{(increase both)}
\]

\[
(x, y) \in r \implies (x - 1, y - 1) \in r \quad \text{(decrease both)}
\]

Which relations satisfy these rules?

- $r = \{(x, y) \mid x = 0 \lor y = 0\}$ ? No
- $r = \{(x, y) \mid x \leq 0 \land 0 \leq y\}$ ? No
- $r = \mathbb{Z} \times \mathbb{Z}$ ? Yes

What is the **smallest** relation (wrt. $\subseteq$)?
Inductively defined relation: example

Define relation $r \subseteq \mathbb{Z} \times \mathbb{Z}$ using these inductive rules.

- $(0, 0) \in r$ (zero)
- $(x, y) \in r \Rightarrow (x, y + 1) \in r$ (increase right)
- $(x, y) \in r \Rightarrow (x + 1, y + 1) \in r$ (increase both)
- $(x, y) \in r \Rightarrow (x - 1, y - 1) \in r$ (decrease both)

Which relations satisfy these rules?

- $r = \{(x, y) | x = 0 \lor y = 0\}$ ? No
- $r = \{(x, y) | x \leq 0 \land 0 \leq y\}$ ? No
- $r = \mathbb{Z} \times \mathbb{Z}$ ? Yes

What is the smallest relation (wrt. $\subseteq$)? $r = \{(x, y) | x \leq y\}$
Example derivation of \((-3, -1) \in r\)

\[
\begin{align*}
(0, 0) \in r \\
(0, 1) \in r \\
(0, 2) \in r \\
(-1, 1) \in r \\
(-2, 0) \in r \\
(-3, -1) \in r
\end{align*}
\]

\[
\begin{align*}
(0, 0) \in r \quad &\text{ (zero)} \\
(x, y) \in r \quad &\text{ (increase right)} \\
(x, y + 1) \in r \\
(x + 1, y + 1) \in r \quad &\text{ (increase both)} \\
(x, y) \in r \quad &\text{ (decrease both)} \\
(x - 1, y - 1) \in r
\end{align*}
\]
Proof that our rules define \{ (x, y) \mid x \leq y \} 

Establish two directions:

- if there exists a derivation, then $x \leq y$
  
  Strategy: induction on derivation, go through each rule

- if $x \leq y$ then there exists a derivation
  
  Strategy (problem-specific): we can find an algorithm that given $x, y$ finds derivation tree (what is the algorithm?)
Proof that our rules define \( \{ (x, y) \mid x \leq y \} \)

Establish two directions:

- if there exists a derivation, then \( x \leq y \)
  Strategy: induction on derivation, go through each rule

- if \( x \leq y \) then there exists a derivation
  Strategy (problem-specific): we can find an algorithm that given \( x, y \) finds derivation tree (what is the algorithm?)

Example: start from \((0, 0)\), then
derive \((0, y - x)\) in \( y - x \) steps of “increase right”,
then depending on whether \( x < 0 \) or \( x > 0 \) apply “increase both” or “decrease both” rule \(|x|\) times.
Inductively defined relations

We can use inductive rules to define type systems, grammars, interpreters, . . .
We define a relation \( r \) using rules of the form

\[
\frac{t_1(\bar{x}) \in r, \ldots, t_n(\bar{x}) \in r}{t(\bar{x}) \in r}
\]

where \( t_i(\bar{x}) \in r \) are assumptions and \( t(\bar{x}) \in r \) is the conclusion. When \( n = 0 \) (no assumptions), the rule is called an axiom.

A derivation tree has nodes marked by tuples \( t(\bar{a}) \) for some specific values \( \bar{a} \) of \( \bar{x} \).
We define relation \( r \) as the set of all tuples for which there exists a derivation tree. This is the smallest relation that satisfies the rules.
Amyrli language

Tiny language similar to one in the project.
Works only on integers and booleans.

(Initial) program is a pair \((e_{top}, t_{top})\) where

- \(e_{top}\) is the top-level environment mapping function names to function definitions
- \(t_{top}\) is the top-level term (expression) that starts execution

Function definition for a given function name is a tuple of:
parameter list \(\bar{x}\), parameter types \(\bar{\tau}\), expression representing function body \(t\), and result type \(\tau_0\).

Expressions are formed by invoking primitive functions \((+, -, \leq, &&)\), invocations of defined functions, or \textbf{if} expressions.
No local \texttt{val} definitions nor \textbf{match}. e will remain fixed
Amyrli: abstract syntax of terms

\[ t := true | false | c_l | f(t_1, \ldots, t_n) | \textbf{if} (t) \ t_1 \ \textbf{else} \ t_2 \]

where

- \( c_l \in \mathbb{Z} \) denotes integer constant
- \( f \) denotes either application of a user-defined function or one of the primitive operators
Program representation as a mathematical structure

\[ p_{\text{fact}} = (e, \text{fact}(2)) \]
where \( e(\text{fact}) = (n, \text{Int}, \text{if } (n \leq 1) \ 1 \ \text{else} \ n \ast \text{fact}(n - 1), \text{Int}) \)
Operational semantics of Amyrli: \texttt{if} expression

We specify the result of executing the program as an inductively defined binary (infix) relation "\(\leadsto\)" on programs. If the top-level expression becomes a constant after some number of steps of \(\leadsto\), we have computed the result: \(t \leadsto c\)

Rules for \texttt{if}:

\[
\begin{align*}
  b & \leadsto b' \\
  \text{(if} \ (b) \ t_1 \text{ else} \ t_2 \text{)} & \leadsto \text{(if} \ (b') \ t_1 \text{ else} \ t_2 \text{)}
\end{align*}
\]

\[
\begin{align*}
  \text{(if} \ (true) \ t_1 \text{ else} \ t_2 \text{)} & \leadsto t_1
\end{align*}
\]

\[
\begin{align*}
  \text{(if} \ (false) \ t_1 \text{ else} \ t_2 \text{)} & \leadsto t_2
\end{align*}
\]
Operational semantics of Amyrli: primitives

Logical operators:

\[
\begin{align*}
&b_1 \leadsto b'_1 \\
& (b_1 \land\land b_2) \leadsto (b'_1 \land\land b_2) \\
& (\text{true} \land\land b_2) \leadsto b_2 \\
& (\text{false} \land\land b_2) \leadsto \text{false}
\end{align*}
\]

Arithmetic:

\[
\begin{align*}
&k_1 \leadsto k'_1 \\
& (k_1 + k_2) \leadsto (k'_1 + k_2) \\
& k_2 \leadsto k'_2 \\
& (c + k_2) \leadsto (c + k'_2) \quad c \in \mathbb{Z}
\end{align*}
\]

\[
(c_1 + c_2) \leadsto c 
\quad c_1, c_2, c \in \mathbb{Z}, \quad c = c_1 + c_2
\]
Operational semantics: user function $f$

If $c_1, \ldots, c_{i-1}$ are constants, then (as expected in call-by-value)

$$\begin{align*}
  t_i & \leadsto t'_i \\
  f(c_1, \ldots, c_{i-1}, t_i, \ldots) & \leadsto f(c_1, \ldots, c_{i-1}, t'_i, \ldots)
\end{align*}$$

Let the environment $e$ define $f$ by $e(f) = ((x_1, \ldots, x_n), \bar{\tau}, tf, \tau_0)$

- $(x_1, \ldots, x_n)$ is the list of formal parameters of $f$
- $tf$ is the body of the function $f$

Then we can apply rule

$$f(c_1, \ldots, c_n) \leadsto tf[x_1 := c_1, \ldots, x_n := c_n]$$

In general, if $t$ is term, then $t[x_1 := t_1, \ldots, x_n := t_n]$ denotes result of substituting (replacing) in $t$ each variable $x_i$ by term $t_i$. 
Execution of factorial example program

\[ p_{\text{fact}} = (e, \text{fact}(2)) \]

where \( e(\text{fact}) = (n, \text{Int}, \text{if } (n \leq 1) \ 1 \ \text{else} \ n \ast \text{fact}(n - 1), \text{Int}) \)

\[ \text{fact}(2) \leadsto \]
Execution of factorial example program

\[ p_{\text{fact}} = (e, \text{fact}(2)) \]
where \[ e(\text{fact}) = (n, \text{Int}, \text{if } (n \leq 1) \ 1 \ \text{else } n \ast \text{fact}(n - 1), \text{Int}) \]

\[ \text{fact}(2) \rightarrow \]
if (2 \leq 1) 1 else 2 \ast \text{fact}(2 - 1) \rightarrow
Execution of factorial example program

\[ p_{\text{fact}} = (e, \text{fact}(2)) \]

where \( e(\text{fact}) = (n, \text{Int}, \text{if } (n \leq 1) 1 \text{ else } n \times \text{fact}(n - 1), \text{Int}) \)

\[ \text{fact}(2) \rightsquigarrow \]

\[ \text{if } (2 \leq 1) 1 \text{ else } 2 \times \text{fact}(2 - 1) \rightsquigarrow \]

\[ \text{if } (\text{false}) 1 \text{ else } 2 \times \text{fact}(2 - 1) \rightsquigarrow \]
Execution of factorial example program

\[ p_{\text{fact}} = (e, \text{fact}(2)) \]
where \( e(\text{fact}) = (n, \text{Int}, \text{if } (n \leq 1) 1 \text{ else } n \times \text{fact}(n - 1), \text{Int}) \)

\[
\begin{align*}
\text{fact}(2) & \rightarrow \\
\text{if } (2 \leq 1) 1 \text{ else } 2 \times \text{fact}(2 - 1) & \rightarrow \\
\text{if } (\text{false}) 1 \text{ else } 2 \times \text{fact}(2 - 1) & \rightarrow \\
2 \times \text{fact}(2 - 1) & \rightarrow
\end{align*}
\]
Execution of factorial example program

\[ p_{\text{fact}} = (e, \text{fact}(2)) \]
where \( e(\text{fact}) = (n, \text{Int}, \text{if } (n \leq 1) 1 \text{ else } n \times \text{fact}(n - 1), \text{Int}) \)

\[
\begin{align*}
\text{fact}(2) & \mapsto \\
\text{if } (2 \leq 1) 1 \text{ else } 2 \times \text{fact}(2 - 1) & \mapsto \\
\text{if } (\text{false}) 1 \text{ else } 2 \times \text{fact}(2 - 1) & \mapsto \\
2 \times \text{fact}(2 - 1) & \mapsto \\
2 \times \text{fact}(1) & \mapsto
\end{align*}
\]
Execution of factorial example program

\[ p_{\text{fact}} = (e, \text{fact}(2)) \]
where \( e(\text{fact}) = (n, \text{Int}, \text{if } (n \leq 1) 1 \text{ else } n \times \text{fact}(n - 1), \text{Int}) \)

\[
\begin{align*}
\text{ fact}(2) & \leadsto \\
\text{ if } (2 \leq 1) 1 \text{ else } 2 \times \text{fact}(2 - 1) & \leadsto \\
\text{ if } (\text{false}) 1 \text{ else } 2 \times \text{fact}(2 - 1) & \leadsto \\
2 \times \text{fact}(2 - 1) & \leadsto \\
2 \times \text{fact}(1) & \leadsto \\
2 \times (\text{if } (1 \leq 1) 1 \text{ else } 1 \times \text{fact}(1 - 1)) & \leadsto 
\end{align*}
\]
Execution of factorial example program

\[ p_{\text{fact}} = (e, \text{fact}(2)) \]
where \( e(\text{fact}) = (n, \text{Int}, \text{if } (n \leq 1) 1 \text{ else } n \times \text{fact}(n - 1), \text{Int}) \)

\[
\text{fact}(2) \leadsto \\
\text{if } (2 \leq 1) 1 \text{ else } 2 \times \text{fact}(2 - 1) \leadsto \\
\text{if } (\text{false}) 1 \text{ else } 2 \times \text{fact}(2 - 1) \leadsto \\
2 \times \text{fact}(2 - 1) \leadsto \\
2 \times \text{fact}(1) \leadsto \\
2 \times (\text{if } (1 \leq 1) 1 \text{ else } 1 \times \text{fact}(1 - 1)) \leadsto \\
2 \times (\text{if } (true) 1 \text{ else } 1 \times \text{fact}(1 - 1)) \leadsto \\
2 \times 1 \leadsto \\
2 \]
Execution of factorial example program

\[ p_{\text{fact}} = (e, \text{fact}(2)) \]
where \( e(\text{fact}) = (n, \text{Int}, \text{if } (n \leq 1) 1 \text{ else } n \ast \text{fact}(n - 1), \text{Int}) \)

\[
\begin{align*}
\text{fact}(2) & \mapsto \\
\text{if } (2 \leq 1) 1 \text{ else } 2 \ast \text{fact}(2 - 1) & \mapsto \\
\text{if } (\text{false}) 1 \text{ else } 2 \ast \text{fact}(2 - 1) & \mapsto \\
2 \ast \text{fact}(2 - 1) & \mapsto \\
2 \ast \text{fact}(1) & \mapsto \\
2 \ast (\text{if } (1 \leq 1) 1 \text{ else } 1 \ast \text{fact}(1 - 1)) & \mapsto \\
2 \ast (\text{if } (\text{true}) 1 \text{ else } 1 \ast \text{fact}(1 - 1)) & \mapsto \\
2 \ast 1 & \mapsto 
\end{align*}
\]
Execution of factorial example program

\[ p_{\text{fact}} = (e, \text{fact}(2)) \]

where \( e(\text{fact}) = (n, \text{Int}, \text{if } (n \leq 1) 1 \text{ else } n \times \text{fact}(n - 1), \text{Int}) \)

\[
\text{fact}(2) \sim \\
\text{if } (2 \leq 1) 1 \text{ else } 2 \times \text{fact}(2 - 1) \sim \\
\text{if } (\text{false}) 1 \text{ else } 2 \times \text{fact}(2 - 1) \sim \\
2 \times \text{fact}(2 - 1) \sim \\
2 \times \text{fact}(1) \sim \\
2 \times (\text{if } (1 \leq 1) 1 \text{ else } 1 \times \text{fact}(1 - 1)) \sim \\
2 \times (\text{if } (\text{true}) 1 \text{ else } 1 \times \text{fact}(1 - 1)) \sim \\
2 \times 1 \sim \\
2 \]
Getting stuck

If a term \( t \) makes no sense, we introduce no rule to define its evaluation, so there is no \( t' \) such that \( t \leadsto t' \)

Example: consider this top-level expression:

\[
\text{if } (5) \text{ else 7}
\]

the expression 5 cannot be evaluated further and is a constant, but there are no rules for when condition of \text{if} \ is a number constant; there are only rules for boolean constants.

Such terms, that are not constants and have no applicable rules, are called \textit{stuck}, because no further steps are possible.

Stuck terms indicate errors. Type checking is a way to detect them \textit{statically}, without trying to (dynamically) execute a program and see if it will get stuck or produce result.
Type Rules: Program

After the definition of operational semantics, we define type rules (also inductively). Given initial program \((e, t)\) define

\[
\Gamma_0 = \{ (f, \tau_1 \times \cdots \times \tau_n \rightarrow \tau_0) \mid (f, -, \tau_1, \ldots, \tau_n, t_f, \tau_0) \in e \}
\]

We say program type checks iff:
(1) the top-level expression type checks:

\[
\Gamma_0 \vdash t : \tau
\]

and

(2) each function body type checks:

\[
\Gamma_0 \oplus \{ (x_1, \tau_1), \ldots, (x_n, \tau_n) \} \vdash t_f : \tau_0
\]

for each \((f, (x_1, \ldots, x_n), (\tau_1, \ldots, \tau_n), t_f, \tau_0) \in e\)
Type Rules are as Usual

\[
\Gamma \vdash b : \text{Bool}, \quad \Gamma \vdash t_1 : \tau, \quad \Gamma \vdash t_2 : \tau \\
\frac{}{\Gamma \vdash (\text{if} \ (b) \ t_1 \ \text{else} \ t_2) : \tau}
\]

\[
\Gamma \vdash f : \tau_1 \times \cdots \times \tau_n \to \tau_0, \quad \Gamma \vdash t_1 : \tau_1, \ldots, \quad \Gamma \vdash t_n : \tau_n \\
\frac{}{\Gamma \vdash f(t_1, \ldots , t_n) : \tau_0}
\]

We treat primitives like applications of functions e.g.

\[+ : \text{Int} \times \text{Int} \to \text{Int}\]

\[\leq : \text{Int} \times \text{Int} \to \text{Bool}\]

\[\&\& : \text{Bool} \times \text{Bool} \to \text{Bool}\]
Soundness through progress and preservation

Soundness theorem: if a program type checks, then its evaluation does not get stuck.

Proof uses the following two lemmas, which is a common approach:

- progress: if a program type checks, it is not stuck: if

\[ \Gamma \vdash t : \tau \]

then either \( t \) is a constant or there exists \( t' \) such that \( t \leadsto t' \)

- preservation: if a program type checks and makes one step, the result again type checks

here: type checks and has the same type: if

\[ \Gamma \vdash t : \tau \]

and \( t \leadsto t' \) then

\[ \Gamma \vdash t' : \tau \]
Proof of progress and preservation - case of if

We prove conjunction of progress and preservation by induction on term $t$ such that $\Gamma \vdash t : \tau$. The operational semantics defines the non-error cases of an interpreter, which enables case analysis. Consider if. By type checking rules, if can only type check if its condition $b$ type checks and has type Bool. By inductive hypothesis and progress either $b$ is constant or it can be reduced to $b'$. If it is constant one of these rules apply:

$$
\frac{}{(\text{if} \ (\text{true}) \ t_1 \ \text{else} \ t_2) \leadsto t_1}
$$

$$
\frac{}{(\text{if} \ (\text{false}) \ t_1 \ \text{else} \ t_2) \leadsto t_2}
$$

and the result, by type rule for if, has type $\tau$. If $b'$ is not constant and the assumption of the rule

$$
\frac{}{b \leadsto b'}
$$

applies so $t$ also makes progress. Moreover, by preservation $b'$ also has type Bool, so the entire expression can be typed as $\tau$ by re-using the type derivations for $t_1$ and $t_2$. 
Progress and preservation - user defined functions

Following the cases of operational semantics, either all arguments of a function have been evaluated to a constant, or some are not yet constant. If they are not all constants, the case is as for the condition of if, and we establish progress and preservation analogously. Otherwise rule

\[
  f(c_1, \ldots, c_n) \leadsto t_f[x_1 := c_1, \ldots, x_n := c_n]
\]

applies, so progress is ensured. For preservation, we need to show

\[
  \Gamma \vdash t_f[x_1 := c_1, \ldots, x_n := c_n] : \tau
\]  \hspace{1cm} (\ast)

where \( e(f) = ((x_1, \ldots, x_n), (\tau_1, \ldots, \tau_n), t_f, \tau_0) \) and \( t_f \) is the body of \( f \). According to type rules \( \tau = \tau_0 \) and \( \Gamma \vdash c_i : \tau_i \).
Function $f$ definition type checks, so $\Gamma' \vdash t_f : \tau_0$ where 
$\Gamma' = \Gamma \oplus \{(x_1, \tau_1), \ldots, (x_n, \tau_n)\}$.

Consider the type derivation tree for $t_f$ and replace each use of 
$\Gamma' \vdash x_i : \tau_i$ with $\Gamma \vdash c_i : \tau_i$. The result is a type derivation for $(\ast)$:

$$
\Gamma \vdash t_f[x_1 := c_1, \ldots, x_n := c_n] : \tau
$$

\((\ast)\)

Therefore, the preservation holds in this case as well.
Function $f$ definition type checks, so $\Gamma' \vdash t_f : \tau_0$ where
$\Gamma' = \Gamma \oplus \{(x_1, \tau_1), \ldots, (x_n, \tau_n)\}$.
Consider the type derivation tree for $t_f$ and replace each use of
$\Gamma' \vdash x_i : \tau_i$ with $\Gamma \vdash c_i : \tau_i$. The result is a type derivation for ($\ast$):

$\Gamma \vdash t_f[x_1 := c_1, \ldots, x_n := c_n] : \tau$  \hspace{1cm} (\ast)

Therefore, the preservation holds in this case as well.

Exercise: prove the above step that replacing variables with
constants of the same type transforms term that has type
derivation with type $\tau$ into a term that again has a derivation
with type $\tau$. Is there a more general statement?