We start our discussion of programming languages with MinHs:

- Haskell-like language
- functional language: no side effects, functions are first class citizens
- call-by-value
- fully typed
- types have to be provided by programmer
Concrete Syntax:

Variables \( id ::= \ldots \)

Integer values \( n ::= \ldots \)

Types \( \tau ::= \text{Bool} \mid \text{Int} \mid \tau_1 \to \tau_2 \mid (\tau) \)

Infix Operators \( \otimes ::= + \mid * \mid - \mid = \)

Exprs \( e ::= id \mid n \mid (e) \mid e_1 \otimes e_2 \mid e_1 \ e_2 \)
\( \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \)
\( \mid \text{letfun } id_1 : (\tau_1 \to \tau_2) \ id_2 = e \)

Note: the definition of the concrete syntax is ambiguous — the usual precedence and associativity rules apply.
letfun \( id_1 :: (\tau_1 \to \tau_2) \) \( id_2 = e \)

Note that

- a function is an expression just like any other
- the scope of the function variable is only its own body
- the function accepts only one argument at a time
letfun $id_1 :: (\tau_1 \to \tau_2) \, id_2 = e$

Note that
- a function is an expression just like any other
- the scope of the function variable is only its own body
- the function accepts only one argument at a time

We could
- allow multiple variables
- add let-bindings

but this would complicate the matter without adding anything interesting
Example program:

```haskell
let fun div5 :: (Int -> Int) x =
  if x < 5 then
    0
  else
    1 + div5 (x - 5)
```
Example program:

```haskell
letfun div:: (Int -> Int -> Int) x =
    letfun divx:: (Int -> Int) y =
        if y < x then
            0
        else
            1 + divx (y-x)
```

So, \texttt{div 5} is the same as \texttt{div5} of the previous slide, that is \texttt{div 5 10} should evaluate to 2.
**FIRST-ORDER ABSTRACT SYNTAX**

- replace infix by postfix operators:
  - $e_1 + e_2$ becomes `plus(e_1, e_2)`
  - `if $e_1$ then $e_2$ else $e_3$` becomes `if(e_1, e_2, e_3)`

- application is explicit:
  - $e_1 e_2$ becomes `apply(e_1, e_2)`

- function definitions:
  - `let fun f :: (τ₁ -> τ₂) x = e` becomes `fun(τ₁, τ₂, f, x, e)`
Higher-order Abstract Syntax

The representation of function definition and let-bindings change to express that binding of the newly introduced variables:

\[ \text{fun}(\tau_1, \tau_2, f.x.e) \]: The scope of \( f \) and \( x \) is \( e \).
STATIC SEMANTICS

Has to check if

➔ all variables are defined
➔ expressions are well typed
STATIC SEMANTICS

Has to check if

→ all variables are defined
→ expressions are well typed

The environment $\Gamma$ has to contain type information:

$$\Gamma = \{ x_1 : \text{Int}, x_2 : \text{Bool}, \ldots \}$$

We still work under the assumption that all variable and function names are unique.
Define the typing rules over the structure of the abstract syntax of MinHs:

We need typing rules for

- constant values, variables
- operators
- function definition
- application
Typing Rules for MinHs

\[
\begin{align*}
x : \tau & \in \Gamma \\
\Gamma \vdash \text{var}(x) : \tau
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{num}(n) : \text{int}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash \text{const}(b) : \text{Bool} \quad b \in \{\text{true}, \text{false}\}
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
\Gamma \vdash \text{plus}(e_1, e_2) : \text{int}
\end{align*}
\]
\[
\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \\
\Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau \\
\]

\[
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \\
\Gamma \vdash \text{apply}(e_1, e_2) : \tau_2 \\
\]

\[
\Gamma \cup \{ f : \tau_1 \rightarrow \tau_2 \} \cup \{ x : \tau_1 \} \vdash e : \tau_2 \\
\Gamma \vdash \text{fun}(\tau_1, \tau_2, f.x.e) : \tau_1 \rightarrow \tau_2 \\
\]
Observations:

- There is only one typing rule for each kind of expression
- The typing is syntax directed
  - form of syntax uniquely identifies typing rule
- As a consequence, inversion principle is applicable
INVERSION

Consider the typing rules for *if*-expressions:

\[
\begin{align*}
\Gamma \vdash e_1 : \text{bool} & \quad \Gamma \vdash e_2 : \tau & \quad \Gamma \vdash e_3 : \tau \\
\hline
\Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau
\end{align*}
\]

It says that:

\[\Rightarrow\] if \( \Gamma \vdash e_1 : \text{bool} \) is derivable, and \( \Gamma \vdash e_2 : \tau \) and \( \Gamma \vdash e_3 : \tau \) is derivable

\[\Rightarrow\] then \( \Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau \) is derivable
Consider the typing rules for if-expressions:

\[
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau}
\]

It says that:

- if \( \Gamma \vdash e_1 : \text{bool} \) is derivable, and \( \Gamma \vdash e_2 : \tau \) and \( \Gamma \vdash e_3 : \tau \) is derivable
- then \( \Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau \) is derivable

However, if

- \( \Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau \) is derivable

we know that this can only be because the above rule was applicable, and therefore

- \( \Gamma \vdash e_1 : \text{bool} \) is derivable, and \( \Gamma \vdash e_2 : \tau \) and \( \Gamma \vdash e_3 : \tau \) is derivable
Therefore, inverse of the rule also holds:

\[
\Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau \\
\Gamma \vdash e_1 : \text{bool}
\]

\[
\Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau \\
\Gamma \vdash e_2 : \tau
\]

\[
\Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau \\
\Gamma \vdash e_3 : \tau
\]

Inversion holds for all the other typing rules of MinHs as well! Formally, this can easily (really!) be proven using rule induction.
Structured operational semantics:

→ Initial states: all well-typed expressions

→ Final states:
  
  • boolean and integer constants
Structured operational semantics:

→ **Initial states:** all well-typed expressions

→ **Final states:**
  - boolean and integer constants
  - and functions!
Structured operational semantics:

- **Initial states:** all well-typed expressions
- **Final states:**
  - boolean and integer constants
  - and functions!

**Evaluation of Built-in Operations:** see evaluation of arithmetic expressions

\[
\text{plus}(\text{num}(n), \text{num}(m)) \rightarrow \text{num}(n + m)
\]

Similarly for the other operations
Evaluation of if-expression:

\[
\begin{align*}
e_1 &\mapsto e'_1 \\
\text{if}(e_1, e_2, e_3) &\mapsto \text{if}(e'_1, e_2, e_3) \\
\text{if}(\text{const}(\text{true}), e_1, e_2) &\mapsto e_1 \\
\text{if}(\text{const}(\text{false}), e_1, e_2) &\mapsto e_2
\end{align*}
\]
Function Application: (we omit the type here)
Function Application: (we omit the type here)

(\texttt{letfun f x = x \* (x + 1)} \ 5

evaluates to

\ 5 \* (5 + 1)
Function Application: (we omit the type here)

```
(letfun f x = x * (x + 1)) 5
```
evaluates to

```
5 * (5 + 1)
```

Is it, in general, enough to replace the variable by the value?
No, not for recursive functions:

\[
(\text{letfun } f \ x = \ \text{if } x < 1 \text{ then } 1 \text{ else } x \times f(x-1)) \ 3
\]
evaluates to

\[
\text{if } 3 < 1 \text{ then } 1 \text{ else } 3 \times f(3-1)
\]
No, not for recursive functions:

(letfun f x = if x<1 then 1 else x*f(x-1)) 3

evaluates to

if 3<1 then
  1
else
  3* f(3-1)

but something is missing, f is now out of scope!
\[(\text{letfun } f \ x = \ \text{if } x < 1 \ \text{then } 1 \ \text{else } x \times f(x - 1)) \ 3\]

We have to replace

→ \(x\) by 3

→ \(f\) by \((\text{letfun } f \ x = \ \text{if } x < 1 \ \text{then } 1 \ \text{else } x \times f(x - 1))\)

if 3<1 then

1

else

3*\((\text{letfun } f \ x = \ \text{if } x < 1 \ \text{then } 1 \ \text{else } x \times f(x - 1))\)(3-1)

fi
Application:

$$e_1 \rightarrow e'_1$$

$$\text{apply}(e_1, e_2) \leftrightarrow \text{apply}(e'_1, e_2)$$
Application:

\[
\begin{align*}
\frac{e_1 \mapsto e'_1}{\text{apply}(e_1, e_2) \mapsto \text{apply}(e'_1, e_2)} \\
\frac{e_2 \mapsto e'_2}{\text{apply}(\text{fun}(\ldots), e_2) \mapsto \text{apply}(\text{fun}(\ldots), e'_2)}
\end{align*}
\]
Application:

\[
\begin{align*}
\frac{e_1 \mapsto e_1'}{
\text{apply}(e_1, e_2) \mapsto \text{apply}(e_1', e_2)}
\end{align*}
\]

\[
\begin{align*}
\frac{e_2 \mapsto e_2'}{
\text{apply}(\text{fun}(\ldots), e_2) \mapsto \text{apply}(\text{fun}(\ldots), e_2')}
\end{align*}
\]

\[
\text{apply}(\text{fun}(\tau_1, \tau_2, f.x.e_1), v) \mapsto \{\text{fun}(\tau_1, \tau_2, f.x.e)/f\}{v/x}e_1
\]
letfun div:: (Int -> Int -> Int) x =
  letfun divx:: (Int -> Int) y =
    if y < x then
      0
    else
      1 + divx (y-x)
apply ( 
    apply ( 
        fun (div.x. (fun divx y. if (less (y,x), ....)) 
            5), 
        7) 
)
apply (apply (fun (div.x. (fun divx y. if (less (y,x), ....)) 5), 7)
    ➞
apply (fun (divx. y. if (less(y,5), 0, plus (1,divx (minus (y,5))))), 7)
apply (apply (fun (div x. (fun div x y. if (less y x, ....)) 5), 7)

⇒

apply (fun (div x y if (less y 5), 0, plus (1, div x (minus y 5))), 7)

⇒

if (less 7 5), 0, plus (1, apply (fun (div x y if .... , (minus (7,5))))))
if (False, 0,
    plus (1, apply (fun (divx y. if .....), (minus (7,5))))
)
if (False, 0,
    plus (1, apply (fun (divx y. if .....), (minus (7,5))))))

⇒

plus (1, apply (fun (divx y. if .....), (minus (7,5))))
if (False, 0,
    plus (1, apply (fun (divx y. if .....), (minus (7,5))))))

\[ \mapsto \]

plus (1, apply (fun (divx y. if .....), (minus (7,5))))

\[ \mapsto \]

plus (1, apply (fun (divx y. if .....), 2)))
if (False, 0,
    plus (1, apply (fun (divx y. if .....), (minus (7,5))))))

⇒

plus (1, apply (fun (divx y. if .....), (minus (7,5))))

⇒

plus (1, apply (fun (divx y. if .....), 2))

⇒

plus (1, if (less(2,5),0,plus (1, fun (.....))))
if (False, 0,
    plus (1, apply (fun (divx y. if .....), (minus (7,5)))))

⇒

plus (1, apply (fun (divx y. if .....), (minus (7,5))))

⇒

plus (1, apply (fun (divx y. if .....), 2))

⇒

plus (1, if (less(2,5),0,plus (1, fun (.....))))

⇒

plus (1, if (True,0,plus (1, fun (.....))))
if (False, 0,
    plus (1, apply (fun (divx y. if .....), (minus (7,5))))
)

⇒

plus (1, apply (fun (divx y. if .....), (minus (7,5))))

⇒

plus (1, apply (fun (divx y. if .....), 2))

⇒

plus (1, if (less(2,5),0,plus (1, fun (....))))

⇒

plus (1, if (True,0,plus (1, fun (....))))

⇒

plus (1, 0)
Properties of MinHs’s Dynamic Semantics

→ is the semantics of MinHs well-defined?

• assigns at most one value to each expression:

\[ \text{if } e \mapsto v \text{ and } e \mapsto v' \text{ then } v = v' \]
Properties of MinHs’s Dynamic Semantics

- Is the semantics of MinHs well-defined?
  - Assigns at most one value to each expression:
    \[ e \mapsto v \text{ and } e \mapsto v' \text{ then } v = v' \]
  - If not, programs would not have a definite meaning
    - For which type of language would this make sense?
Languages like Java, Haskell, MinHs are type safe
Languages like Java, Haskell, MinHs are type safe

What exactly does it mean?
Languages like Java, Haskell, MinHs are type safe

What exactly does it mean?

- a statement about the relation between static and dynamic semantics
- type-safety gives us some guarantees about the dynamic behaviour of type correct programs
Languages like Java, Haskell, MinHs are type safe

What exactly does it mean?

- a statement about the relation between static and dynamic semantics
- type-safety gives us some guarantees about the dynamic behaviour of type correct programs

In our framework, type safety amounts to the following two properties:

- Preservation
- Progress
Preservation:

Evaluation does not change the type of an expression:

If $\vdash e : \tau$, and $e \mapsto e'$, then $\vdash e' : \tau$
Preservation:

Evaluation does not change the type of an expression:

\[
\text{If } \vdash e : \tau, \text{ and } e \mapsto e', \text{ then } \vdash e' : \tau
\]

Progress:

A well-typed program cannot get stuck:

\[
\text{If } e \text{ is well-typed, then either } e \text{ is a final state, or there exists } e', \text{ with } e \mapsto e'
\]
For progress and preservation to hold, we have to know something about the primitive operation of the abstract machine:

→ **Preservation:**
  - given a term \( o(v_1, \ldots, v_n) : \tau \), and applying the corresponding abstract machine operation \( o \) to \( v_1 \) to \( v_n \) yields a value \( v \), then \( v : \tau \)
  - if \( \Gamma \vdash e : \tau \), \( \Gamma \vdash x : \tau' \) and \( \Gamma \vdash v : \tau' \), then \( \{v/x\}e : \tau \)

→ **Progress:**
  - given a term \( o(v_1, \ldots, v_n) : \tau \), then applying the corresponding abstract machine operation \( o \) to \( v_1 \) to \( v_n \) yields a value for all possible \( v_i \)'s.
For progress and preservation to hold, we have to know something about the primitive operation of the abstract machine:

→ **Preservation:**

- given a term $o(v_1, \ldots, v_n) : \tau$, and applying the corresponding abstract machine operation $o$ to $v_1$ to $v_n$ yields a value $v$, then $v : \tau$
- if $\Gamma \vdash e : \tau$, $\Gamma \vdash x : \tau'$ and $\Gamma \vdash v : \tau'$, then $\{v/x\}e : \tau$

→ **Progress:**

- given a term $o(v_1, \ldots, v_n) : \tau$, then applying the corresponding abstract machine operation $o$ to $v_1$ to $v_n$ yields a value for all possible $v_i$'s.

We can show that progress and preservation holds for MinHs?
For progress and preservation to hold, we have to know something about the primitive operation of the abstract machine:

→ **Preservation:**

- given a term \( o(v_1, \ldots, v_n) : \tau \), and applying the corresponding abstract machine operation \( o \) to \( v_1 \) to \( v_n \) yields a value \( v \), then \( v : \tau \)

- if \( \Gamma \vdash e : \tau \), \( \Gamma \vdash x : \tau' \) and \( \Gamma \vdash v : \tau' \), then \( \{v/x\}e : \tau \)

→ **Progress:**

- given a term \( o(v_1, \ldots, v_n) : \tau \), then applying the corresponding abstract machine operation \( o \) to \( v_1 \) to \( v_n \) yields a value for all possible \( v_i \)'s.

We can show that progress and preservation holds for MinHs?

→ only if we exclude division
Rule Induction over the evaluation rules: We show that for every SOS evaluation rule, this evaluation step preserves the type for a well typed expression (if the step in the premise preserves the type)

1. Addition of two values:

\[
\text{plus}(\text{num}(n), \text{num}(m)) \leftrightarrow \text{num}(n + m)
\]

For all \( \Gamma, \Gamma \vdash \text{plus}(\text{num}(n)), \text{num}(m) : \text{Int} \), and
\( \Gamma \vdash \text{num}(n + m) : \text{Int} \), therefore evaluation step preserves type.

2. Addition of value and expression:

\[
\frac{e \leftrightarrow e'}{
\text{plus}(\text{num}(n), e) \leftrightarrow \text{plus}(\text{num}(n), e')}
\]

I.H.: \( e \leftrightarrow e' \) preserves the type
Rule Induction over the typing rules:

We show for each expression which is well typed, it is either fully evaluated or one of the evaluation rules apply:

➔ In case the expression is a value (boolean, integer or function), claim is trivially true
➔ expression can not be a variable, since states include only closed expressions
if-expression:

\[
\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau \\
\Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau
\]

- I.H.: either \(e_1\) is a value, or there is \(e'_1\), such that \(e_1 \leftrightarrow e'_1\)

Two cases have to be considered:

- If \(e_1\) is a value, it has to have type \(\text{Bool}\) (would not be well-typed otherwise), and therefore be either \(\text{False}\) or \(\text{True}\). Therefore \(\text{if}(e_1, e_2, e_3) \leftrightarrow e_2\) or \(\text{if}(e_1, e_2, e_3) \leftrightarrow e_3\).

- Otherwise, there is a \(e'_1\) with \(e_1 \leftrightarrow e'_1\) (I.H.), and \(\text{if}(e_1, e_2, e_3) \leftrightarrow \text{if}(e'_1, e_2, e_3)\)
if-expression:

\[
\frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \text{if}(e_1, e_2, e_3) : \tau}
\]

- I.H. : either \( e_1 \) is a value, or there is \( e'_1 \), such that \( e_1 \leftrightarrow e'_1 \)

Two cases have to be considered:

- If \( e_1 \) is a value, it has to have type \( \text{Bool} \) (would not be well-typed otherwise), and therefore be either \( \text{False} \) or \( \text{True} \). Therefore \( \text{if}(e_1, e_2, e_3) \leftrightarrow e_2 \) or \( \text{if}(e_1, e_2, e_3) \leftrightarrow e_3 \).

- Otherwise, there is a \( e'_1 \) with \( e_1 \leftrightarrow e'_1 \) (I.H.), and

  \( \text{if}(e_1, e_2, e_3) \leftrightarrow \text{if}(e'_1, e_2, e_3) \)

detailed proof in textbook, pages 58-61
Run-time Errors and Safety

- In a type safe language, stuck states correspond to ill-defined programs
  - treat integer value as pointer
  - out-of-bound access of an array
- Unsafe languages do not have stuck states
  - Something happens, but may not be predictable
What happens if we add division to MinHs?

**Problem:** How can we deal with division by zero?

\[
\begin{align*}
\Gamma &\vdash e_1 : \tau &\Gamma &\vdash e_2 : \tau \\
\hline
\Gamma &\vdash \text{div}(e_1, e_2) : \tau
\end{align*}
\]

The expression \(5/0\) is well-typed, but does not evaluate to a value.
1. **Change static semantics**: can we enhance the type system to check for division by zero?
① **Change static semantics:** can we enhance the type system to check for division by zero?

② **Change dynamic semantics:** can we modify the dynamic semantics such that division by zero does not lead to a stuck state?
Change static semantics: can we enhance the type system to check for division by zero?

In general, such a type system would not be decidable!

Change dynamic semantics: can we modify the dynamic semantics such that division by zero does not lead to a stuck state?
Change static semantics: can we enhance the type system to check for division by zero?
In general, such a type system would not be decidable!

Change dynamic semantics: can we modify the dynamic semantics such that division by zero does not lead to a stuck state?
approach is widely used for type-safe languages
We add a new special expression \texttt{error} to represent run-time errors in the language

\begin{itemize}
  \item What is the type of \texttt{error}?
  \item How does the dynamic semantics deal with \texttt{error}?
\end{itemize}
Division by zero evaluates to `error`:

\[
\text{div}(v, \text{num}(0)) \leftrightarrow \text{error}
\]

As soon as an error is encountered, the computation is interrupted:

\[
\text{plus}(\text{error}, e_2) \leftrightarrow \text{error}
\]

\[
\text{plus}(e_1, \text{error}) \leftrightarrow \text{error}
\]

\[
\text{if}(\text{error}, e_2, e_3) \leftrightarrow \text{error}
\]

and so on
Typing rule for error:

\[ \Gamma \vdash \text{error} : \tau \]

A run-time error can have any type!
Typing rule for `error`:

\[ \Gamma \vdash error : \tau \]

A run-time error can have any type!

What kind of situations lead to checked run-time errors in Haskell?
Now we can restate the type safety property:

If an expression is well-typed, it can only evaluate to a value or evaluate to \texttt{error}. It cannot get stuck in an ill-defined state.