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: \( e ::= id \mid n \mid (e) \mid e_1 \otimes e_2 \mid e_1, e_2 \mid \text{if} e_1 \text{then} e_2 \text{else} e_3 \mid \text{letfun} id_1 :: (\tau_1 \to \tau_2) \text{ id}_2 = e \)

Note: the definition of the concrete syntax is ambiguous — the usual precedence and associativity rules apply.

Letfun:\[\text{letfun } id_1 :: (\tau_1 \to \tau_2) \text{ id}_2 = e \]

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:

\[\text{letfun div5 :: (Int -> Int) x =} \]
\[\quad \text{if } x < 5 \text{ then} \]
\[\quad \quad 0 \]
\[\quad \text{else} \]
\[\quad 1 + \text{div5} (x - 5) \]
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, div 5 is the same as div 5 of the previous slide, that is div 5 10 should evaluate to 2.

**First-order Abstract Syntax**

- replace infix by postfix operators:
  - $e_1 + e_2$ becomes `plus(e1, e2)`
  - if $e_1$ then $e_2$ else $e_3$ becomes `if(e1, e2, e3)`
- application is explicit:
  - $e_1 e_2$ becomes `apply(e1, e2)`
- function definitions:
  - `letfun 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:

```haskell
fun (τ1, τ2, 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

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

\[
\Gamma \vdash x : \tau \\
\Gamma \vdash \text{var}(x) : \tau
\]

\[
\Gamma \vdash \text{var}(x) : \tau
\]

\[
\Gamma \vdash \text{num}(n) : \text{int}
\]

\[
\Gamma \vdash \text{const}(b) : \text{Bool} \\
b \in \{\text{true}, \text{false}\}
\]

\[
\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}
\]

\[
\Gamma \vdash \text{plus}(e_1, e_2) : \text{int}
\]

\[
\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau
\]

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

\[
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1
\]

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

\[
\Gamma \vdash e_1 : \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1
\]

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

\[
\Gamma \cup \{f : \tau_1 \rightarrow \tau_2\} \cup \{x : \tau_1 \vdash e : \tau_2\}
\]

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

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, rule inversion is easy: all rules can be read backwards.
**Example**

Consider the typing rules for if-expressions:

\[
\begin{align*}
Γ ⊢ e₁ : \text{bool} & \quad Γ ⊢ e₂ : τ & \quad Γ ⊢ e₃ : τ \\
\hline
Γ ⊢ \text{if}(e₁, e₂, e₃) : τ
\end{align*}
\]

Therefore, inverse of the rule also holds:

\[
\begin{align*}
Γ ⊢ \text{if}(e₁, e₂, e₃) : τ \\
Γ ⊢ e₁ : \text{bool} \\
Γ ⊢ e₂ : τ \\
Γ ⊢ e₃ : τ
\end{align*}
\]

---

**Dynamic Semantics of MinHs**

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:**

\[
\begin{align*}
e₁ \rightarrow e'_₁ \\
\text{if}(e₁, e₂, e₃) \rightarrow \text{if}(e'_₁, e₂, e₃)
\end{align*}
\]

\[
\begin{align*}
\text{if}(\text{const(true)}, e₁, e₂) \rightarrow e₁ \\
\text{if}(\text{const(false)}, e₁, e₂) \rightarrow e₂
\end{align*}
\]

---

**Function Application:** (we omit the type here)

\[
\text{(letfun f x = x * (x + 1)) 5}
\]

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

\[
\begin{cases}
    \text{if } 3 < 1 \ \text{then} \\
    1 \\
    \text{else} \\
    3 \times f(3-1)
\end{cases}
\]

but something is missing, \(f\) is now out of scope!

We have to replace

\[
\rightarrow x \text{ by } 3
\]

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

The application:

\[
\begin{align*}
&e_1 \mapsto e'_1 \\
&\text{apply}(e_1, e_2) \mapsto \text{apply}(e'_1, e_2)
\end{align*}
\]

\[
\begin{align*}
&e_2 \mapsto e'_2 \\
&\text{apply}(\text{fun}(\ldots), e_2) \mapsto \text{apply}(\text{fun}(\ldots), e'_2)
\end{align*}
\]

\[
\text{apply}(\text{fun}(r_1, r_2, f, x, e_1), v) \mapsto \{\text{fun}(r_1, r_2, f, x, e_1)/f\}[v/x]e_1
\]

\[
\text{letfun div:: (Int -> Int -> Int) x =}
\]

\[
\text{letfun divx:: (Int -> Int) y =}
\]

\[
\begin{cases}
    \text{if } y < x \ \text{then} \\
    0 \\
    \text{else} \\
    1 + \text{divx}(y-x)
\end{cases}
\]
apply (apply (fun (div.x. (fun divx y. if (less (y,x), ....)) 5), 7)
    →
apply (fun (div. y. if (less(y,5), 0, plus (1,divx (minus (y,5)))))), 7)
    →
if (less(7,5), 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)))) →
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:
  if e → v and e → v' then v = v'

• if not, programs would not have a definite meaning
  - for which type of language would this make sense?

Type Safety of MinHs

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:

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

Progress:

A well-typed program cannot get stuck:

\[ \text{If } e \text{ is well-typed, then either } e \text{ is a final state, or there exists } e', \text{ with } e \rightarrow e' \]

For progress and preservation to hold, we have to know something about the primitive operation of the abstract machine:

\[ \Rightarrow \text{Preservation:} \]
- 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 \( \Gamma \vdash e \rightarrow \tau' \), then \( \{v/x\} e : \tau \)

\[ \Rightarrow \text{Progress:} \]
- 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 \( v' \)s.

We can show that progress and preservation holds for MinHs?

\[ \Rightarrow \text{only if we exclude division} \]

---

**Preservation**

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:

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

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

2. Addition of value and expression:

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

I.H.: \( e \rightarrow e' \) preserves the type

---

**Progress**

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

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

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

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

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 if(\( e_1, e_2, e_3 \)) \( \rightarrow e_2 \) or if(\( e_1, e_2, e_3 \)) \( \rightarrow e_3 \).

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

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?

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

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?
   - In general, such a type system would not be decidable!
2. 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
**Checked Errors**

We add a new special expression `error` to represent run-time errors in the language.

- What is the type of `error`?
- How does the dynamic semantics deal with `error`?

---

Division by zero evaluates to `error`:

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

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

\[
\begin{align*}
\text{plus}(\text{error}, e_2) & \rightarrow \text{error} \\
\text{plus}(e_1, \text{error}) & \rightarrow \text{error} \\
\text{if}(\text{error}, e_2, e_3) & \rightarrow \text{error}
\end{align*}
\]

and so on.

---

Typing rule for `error`:

\[
\forall t. \text{error} : t
\]

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 `error`. It cannot get stuck in an ill-defined state.