Typing Example: Number

{ } ⊢ 5 : int

Each

E ⊢ e : T

is a call to type-of-expression with arguments e and E where the result is T
Typing Example: Sum

\[
\begin{align*}
\{\} & \vdash 1 : \text{int} & \{\} & \vdash 2 : \text{int} \\
\{\} & \vdash +(1,2) : \text{int}
\end{align*}
\]

• Actually, the type checker treats primitives like functions, but it could be checked directly as above.

• The above strategy is a good one for HW7, because primitive checking is different than function checking.
Typing Example: Function

\[
\begin{align*}
\{x : \text{int}\} & \vdash x : \text{int} \\
\{x : \text{int}\} & \vdash 2 : \text{int} \\
\{x : \text{int}\} & \vdash + (x, 2) : \text{int} \\
\{\} & \vdash \text{proc}(\text{int } x) + (x, 2) : (\text{int} \to \text{int})
\end{align*}
\]
Typing Example: Function Call

\[
\begin{align*}
\{ \text{x : int} \} & \vdash \text{x : int} \\
\{ \} & \vdash \text{proc(int x)x : (int \rightarrow int)} \\
\{ \} & \vdash 12 : \text{int} \\
\{ \} & \vdash (\text{proc(int x)x 12}) : T_2 \\
(\text{int \rightarrow int}) & = (\text{int \rightarrow T}_2) \\
\text{simplified: int}
\end{align*}
\]

- For inference, create a new type variable for each application
Typing Example: ? Argument

\[
\begin{align*}
\{x : T_1\} \vdash x : T_1 & \quad \{x : T_1\} \vdash 2 : \text{int} \\
\{x : T_1\} \vdash +(x, 2) : \text{int} \\
\{\} \vdash \text{proc(? x)} +(x, 2) : (T_1 \rightarrow \text{int})
\end{align*}
\]

\[T_1 = \text{int}\]

simplified: \((\text{int} \rightarrow \text{int})\)

• Create a new type variable for each ?
Typing Example: ? Argument

\[
\begin{align*}
\{x : T_1\} &\vdash x : T_1 & \{x : T_1\} &\vdash 2 : \text{int} & \{x : T_1\} &\vdash 3 : \text{int} \\
\{x : T_1\} &\vdash \text{if } x \text{ then } 2 \text{ else } 3 : \text{int} \\
\} &\vdash \text{proc}(\ ? \ x) \text{ if } x \text{ then } 2 \text{ else } 3 : (T_1 \rightarrow \text{int})
\end{align*}
\]

\[T_1 = \text{bool}\]

simplified: \((\text{bool} \rightarrow \text{int})\)
Typing Example: Function-Calling Function

\[
\begin{align*}
\{ f : T_1 \} &\vdash f : T_1 \\
\{ f : T_1 \} &\vdash 12 : \text{int} \\
\{ f : T_1 \} &\vdash (f 12) : T_2 \\
\{ \} &\vdash \text{proc}(? f)(f 12) : (T_1 \rightarrow T_2)
\end{align*}
\]

\[
T_1 = (\text{int} \rightarrow T_2)
\]

simplified: \((\text{int} \rightarrow T_2) \rightarrow T_2)\)
Typing Example: Identity

\[
\begin{align*}
\{ x : T_1 \} & \vdash x : T_1 \\
\{ \} & \vdash \texttt{proc(? x) x : (T_1 \rightarrow T_1)}
\end{align*}
\]

\emph{no simplification possible}
Typing Example: Identity Applied

\[ \{ \ x : T_1 \ \} \vdash x : T_1 \]

\[ \{ \ \} \vdash \text{proc}(? \ x) \ x : (T_1 \rightarrow T_1) \]

\[ \{ \ \} \vdash \text{false} : \text{bool} \]

\[ \{ \ \} \vdash (\text{proc}(? \ x)x \ \text{false}) : T_2 \]

\[ (T_1 \rightarrow T_1) = (\text{bool} \rightarrow T_2) \]

simplified: \text{bool}
Typing Example: Function-Making Function

\[
\begin{align*}
\{ x : T_1, y : T_2 \} \vdash x : T_1 \\
\{ x : T_1 \} \vdash \text{proc}(? y) x : (T_2 \rightarrow T_1) \\
\{ \} \vdash \text{proc}(? x) \text{proc}(? y) x : (T_1 \rightarrow (T_2 \rightarrow T_1))
\end{align*}
\]

\textit{no simplification possible}
Typing Example: Compound Primitive Data

\[
\begin{align*}
\{\} & \vdash 1 : \text{int} \\
\{\} & \vdash 2 : \text{int} \\
\{\} & \vdash \text{cons}(1, 2) : [\text{int} : \text{int}] \\
\end{align*}
\]

- In general, \([T_1 : T_2]\) means a pair whose first element is of type \(T_1\) and second element is of type \(T_2\).

- More conventional notation is \((T_1 \times T_2)\).
Typing Example: Compound Primitive Data

\[
\begin{align*}
\{\} & \vdash 1 : \text{int} & \{\} & \vdash 2 : \text{int} \\
\{\} & \vdash \text{cons}(1, 2) : [\text{int} : \text{int}]
\end{align*}
\]

General rule:

\[
\begin{align*}
E & \vdash e_1 : T_1 & E & \vdash e_2 : T_2 \\
E & \vdash \text{cons}(e_1, e_2) : [T_1 : T_2]
\end{align*}
\]
Typing Example: Compound Primitive Data

\[ \{ \} \vdash \text{cons}(1,2) : [\text{int} : \text{int}] \]
\[ \{ \} \vdash \text{car}(\text{cons}(1,2)) : \text{int} \]

General rule:

\[
E \vdash e : [T_1 : T_2] \\
E \vdash \text{car}(e) : T_1 \\
E \vdash e : [T_1 : T_2] \\
E \vdash \text{cdr}(e) : T_2
\]
Infinite Loops

What if we extend the language with a special expression that loops forever?

- if true then 1 else →→ 1
- if false then 1 else →→ loops forever
- if true then proc(? x)x else →→ proc(? x)x

What is the type of ?

For HW7, it's int, but more generally...
Typing Example: Infinite Loop

\[ \{ \} \vdash \text{true : bool} \quad \{ \} \vdash 1 : \text{int} \quad \{ \} \vdash , : T_1 \]

\[ \{ \} \vdash \text{if true then 1 else , : int} \]

\[ T_1 = \text{int} \]

• Create a new type variable for each ,
Type Inference Summary

- New type variable for each `?`
- New type variable for each application
- New type variable for each `,`
- Checking a type equation can force a type variable to match a certain type
The Universe of Programs

- The goal of type-checking is to rule out bad programs
  \[ +\langle 1, \text{true} \rangle \]
- Unfortunately, some good programs will be ruled out, too
  \[ +\langle 1, \text{if true then 1 else false} \rangle \]
Every program falls into one of three categories:

- Programs that run forever
- Programs that crash
- Programs that produce values

The Universe of Programs
The Universe of Programs

- programs that run forever
- programs that crash
- programs that produce values
- well-typed programs

• The idea is that a type checker rules out the error category
The Universe of Programs

- programs that run forever
- programs that crash
- programs that produce values
- well-typed programs

- But a type checker for most languages will allow some errors!

\[ \frac{1}{0} \rightarrow \text{divide by zero} \]
The Universe of Programs

- programs that run forever
- programs that crash on variants
- programs that crash on types
- programs that produce values
- well-typed programs

- Still, a type checker _always_ rules out a certain class of errors
  - Division by 0 is a **variant error**
The Universe of Programs

- programs that run forever
- programs that crash
- programs that produce values
- well-typed programs

- Our language happens to have no variant errors, so the type checker rules out all errors
The Universe of Programs

- programs that run forever
- programs that produce values
- well-typed programs
- programs that crash

• In fact, if we get rid of `letrec`, then every well-typed program terminates with a value!
Intuition for Termination

Recall that to get rid of `letrec`

\[
\text{letrec int sum} = \text{proc(int x)} \\
\quad \text{if zero?(x)} \\
\quad \quad \text{then 0} \\
\quad \quad \text{else } (x, (\text{sum } -(x, 1))) \\
\quad \text{in (sum 10)}
\]

we can use self-application:

\[
\text{let sum} = \text{proc(int x, ? sum)} \\
\quad \text{if zero?(x)} \\
\quad \quad \text{then 0} \\
\quad \quad \text{else } (x, ((\text{sum sum}) -(x, 1))) \\
\quad \text{in ((sum sum) 10)}
\]
Intuition for Termination

But we've already seen that we can't type self-application:

\[
\text{proc}(?_1 \ x)(x \ x)
\]

\[
T_1 \quad T_1
\]

\textit{no type: }T_1 \text{ can't be } (T_1 \to T_2)

The only way around this restriction is to restore \texttt{letrec} or extend the type language.

(Extending the type language in this direction is beyond the scope of the course.)
The Universe of Programs

- There are other ways that we'd like to expand the set of well-formed programs.
The Universe of Programs

- There are other ways that we'd like to expand the set of well-formed programs

- Adjusting the type rules can allow more programs

- Programs that run forever
- Programs that produce values
- Well-typed programs
- Programs that crash
Polymorphism

$$\text{proc}(?_1 y)y$$

$$\text{T}_1$$

$$(\text{T}_1 \to \text{T}_1)$$

let $f = \text{proc}(?_1 y)y : (\text{T}_1 \to \text{T}_1)$

in $\text{if } (f \text{ true}) \text{ then } (f \text{ 1}) \text{ else } (f \text{ 0})$

$$\text{T}_1 \to \text{T}_1$$

$$(\text{T}_1 \to \text{T}_1)$$

$$(\text{T}_1 \to \text{T}_1)$$

$$(\text{T}_1 \to \text{T}_1)$$

*no type: $\text{T}_1$ can't be both $\text{bool}$ and $\text{int}$*
Polymorphism

• New rule: when type-checking the use of a let-bound variable, create fresh versions of unconstrained type variables

\[
\text{let } f = \text{proc}(?_1 y) y : (T_1 \rightarrow T_1) \\
\quad \text{in if } (f \text{ true}) \text{ then } (f \ 1) \text{ else } (f \ 0) \\
\quad (T_2 \rightarrow T_2) \quad (T_3 \rightarrow T_3) \quad (T_4 \rightarrow T_4) \\
\quad \text{int} \\
\text{T}_2 = \text{bool} \quad \text{T}_3 = \text{int} \quad \text{T}_4 = \text{int}
\]

• This rule is called \textit{let-based polymorphism}