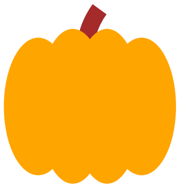
{ } ⊢  : **pumpkin**   { } ⊢  : **face**

{ } ⊢  : **jack-o-lantern**

Typing Example: Number

$$\{\,\} \vdash 5 : \texttt{int}$$

Each

$$E \vdash e : T$$

is a call to **`type-of-expression`** with arguments *e* and *E* where the result is *T*

Typing Example: Sum

$$\frac{\{\,\} \vdash 1 : \texttt{int} \qquad \{\,\} \vdash 2 : \texttt{int}}{\{\,\} \vdash +(1,2) : \texttt{int}}$$

- Actually, the type checker treats primitives like functions, but it could be checked directly as above
- Since the toy language has only single-argument functions, but it has two binary primitives, the above strategy is a good one for HW8

Typing Example: Function

$$\frac{\dfrac{\{\,\mathbf{x} : \texttt{int}\,\} \vdash \mathbf{x} : \texttt{int} \qquad \{\,\mathbf{x} : \texttt{int}\,\} \vdash 2 : \texttt{int}}{\{\,\mathbf{x} : \texttt{int}\,\} \vdash +(\mathbf{x},2) : \texttt{int}}}{\{\,\} \vdash \mathbf{proc}(\mathbf{int}\ \mathbf{x})\ +(\mathbf{x},2) : (\texttt{int} \rightarrow \texttt{int})}$$

Typing Example: Function Call

$$\dfrac{\dfrac{\{\,\mathbf{x} : \texttt{int}\,\} \vdash \mathbf{x} : \texttt{int}}{\{\,\} \vdash \textbf{proc(int x)x} : (\texttt{int} \to \texttt{int})} \qquad \{\,\} \vdash 12 : \texttt{int}}{\{\,\} \vdash (\textbf{proc(int x)x}\ \ 12) : \mathbf{T}_2}$$

$$(\texttt{int} \to \texttt{int}) = (\texttt{int} \to \mathbf{T}_2)$$

simplified: `int`

- Create a new type variable for each application
- We'll see why this is convenient soon...

Typing Example: ? Argument

$$\dfrac{\dfrac{\{\,\mathbf{x} : \mathbf{T}_1\,\} \vdash \mathbf{x} : \mathbf{T}_1 \qquad \{\,\mathbf{x} : \mathbf{T}_1\,\} \vdash 2 : \texttt{int}}{\{\,\mathbf{x} : \mathbf{T}_1\,\} \vdash +(\mathbf{x},2) : \texttt{int}}}{\{\,\} \vdash \textbf{proc}(?\ \mathbf{x})\ +(\mathbf{x},2) : (\mathbf{T}_1 \to \texttt{int})}$$

$$\mathbf{T}_1 = \texttt{int}$$

simplified: `(int → int)`

- Create a new type variable for each ?

Typing Example: ? Argument

$$\dfrac{\dfrac{\{\,\mathbf{x} : \mathbf{T}_1\,\} \vdash \mathbf{x} : \mathbf{T}_1 \qquad \{\,\mathbf{x} : \mathbf{T}_1\,\} \vdash 2 : \texttt{int} \qquad \{\,\mathbf{x} : \mathbf{T}_1\,\} \vdash 3 : \texttt{int}}{\{\,\mathbf{x} : \mathbf{T}_1\,\} \vdash \textbf{if x then } 2 \textbf{ else } 3 : \texttt{int}}}{\{\,\} \vdash \textbf{proc}(?\ \mathbf{x})\ \textbf{if x then } 2 \textbf{ else } 3 : (\mathbf{T}_1 \to \texttt{int})}$$

$$\mathbf{T}_1 = \texttt{bool}$$

simplified: `(bool → int)`

Typing Example: Function-Calling Function

$$\dfrac{\dfrac{\{\,\mathbf{f} : \mathbf{T}_1\,\} \vdash \mathbf{f} : \mathbf{T}_1 \qquad \{\,\mathbf{f} : \mathbf{T}_1\,\} \vdash 12 : \texttt{int}}{\{\,\mathbf{f} : \mathbf{T}_1\,\} \vdash (\mathbf{f}\ 12) : \mathbf{T}_2}}{\{\,\} \vdash \textbf{proc}(?\ \mathbf{f})(\mathbf{f}\ 12) : (\mathbf{T}_1 \to \mathbf{T}_2)}$$

$$\mathbf{T}_1 = (\texttt{int} \to \mathbf{T}_2)$$

simplified: $((\texttt{int} \to \mathbf{T}_2) \to \mathbf{T}_2)$

Typing Example: Identity

$$\frac{\{\, \mathbf{x} : \mathbf{T_1} \,\} \vdash \mathbf{x} : \mathbf{T_1}}{\{\,\} \vdash \mathbf{proc}(?\ \mathbf{x})\ \mathbf{x} : (\mathbf{T_1} \rightarrow \mathbf{T_1})}$$

no simplification possible

Typing Example: Identity Applied

$$\frac{\dfrac{\{\, \mathbf{x} : \mathbf{T_1} \,\} \vdash \mathbf{x} : \mathbf{T_1}}{\{\,\} \vdash \mathbf{proc}(?\ \mathbf{x})\ \mathbf{x} : (\mathbf{T_1} \rightarrow \mathbf{T_1})} \qquad \{\,\} \vdash \mathbf{false} : \texttt{bool}}{\{\,\} \vdash (\mathbf{proc}(?\ \mathbf{x})\mathbf{x}\ \ \mathbf{false}) : \mathbf{T_2}}$$

$$(\mathbf{T_1} \rightarrow \mathbf{T_1}) = (\texttt{bool} \rightarrow \mathbf{T_2})$$

simplfied: `bool`

Typing Example: Function-Making Function

$$\frac{\dfrac{\{\, \mathbf{x} : \mathbf{T_1}, \mathbf{y} : \mathbf{T_2} \,\} \vdash \mathbf{x} : \mathbf{T_1}}{\{\, \mathbf{x} : \mathbf{T_1} \,\} \vdash \mathbf{proc}(?\ \mathbf{y})\ \mathbf{x} : (\mathbf{T_2} \rightarrow \mathbf{T_1})}}{\{\,\} \vdash \mathbf{proc}(?\ \mathbf{x})\ \mathbf{proc}(?\ \mathbf{y})\ \mathbf{x} : (\mathbf{T_1} \rightarrow (\mathbf{T_2} \rightarrow \mathbf{T_1}))}$$

no simplification possible

Infinite Loops

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

- **if true then** 1 **else** Ω $\rightarrow\rightarrow$ 1
- **if false then** 1 **else** Ω $\rightarrow\rightarrow$ *loops forever*
- **if true then proc**(? **x**)**x else** Ω $\rightarrow\rightarrow$ **proc**(? **x**)**x**

What is the type of Ω ?

Typing Example: Infinite Loop

$$\frac{\{\,\}\vdash \textbf{true} : \texttt{bool} \qquad \{\,\}\vdash 1 : \texttt{int} \qquad \{\,\}\vdash \Omega : \mathbf{T}_1}{\{\,\}\vdash \textbf{if true then } 1 \textbf{ else } \Omega : \texttt{int}}$$

$$\mathbf{T}_1 = \texttt{int}$$

- Create a new type variable for each Ω

Type Inference Summary

- New type variable for each ?
- New type variable for each application
- New type variable for each Ω
- Checking a type equation can force a type variable to match a certain type

The Universe of Programs

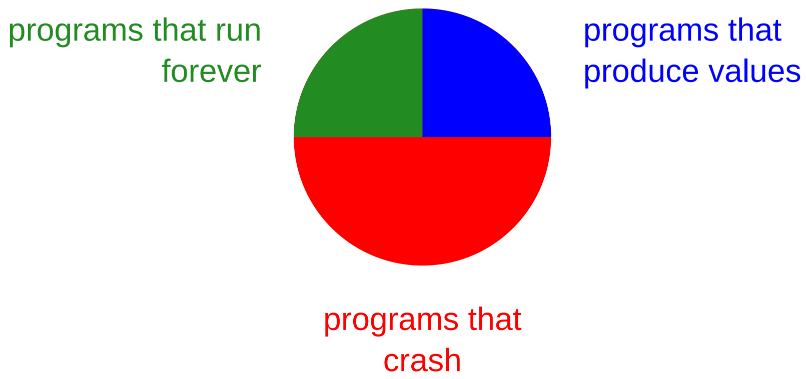
- The goal of type-checking is to rule out bad programs

  +(1, **true**)

- Unfortunately, some good programs will be ruled out, too

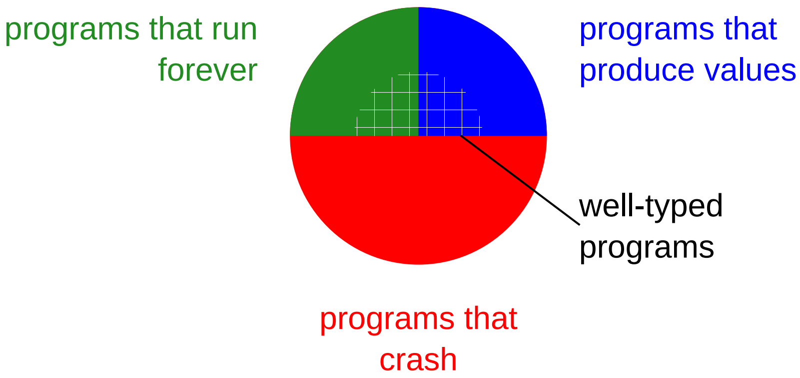
  +(1, **if true then** 1 **else false**)

The Universe of Programs



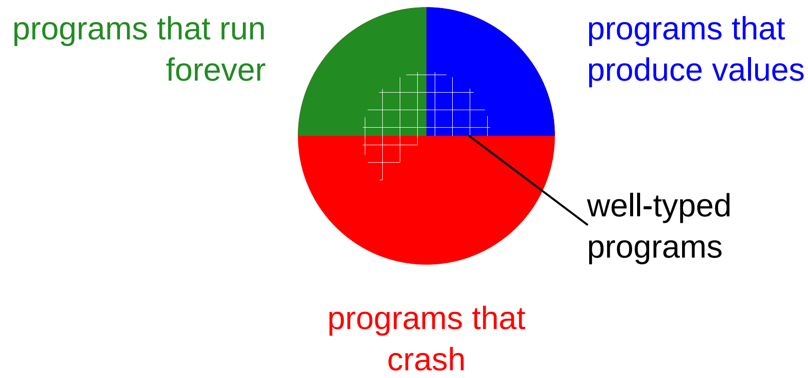
- Every program falls into one of three categories

The Universe of Programs



- The idea is that a type checker rules out the error category

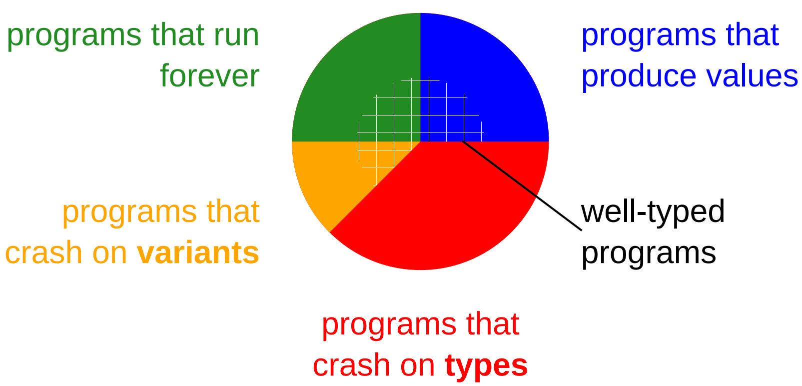
The Universe of Programs



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

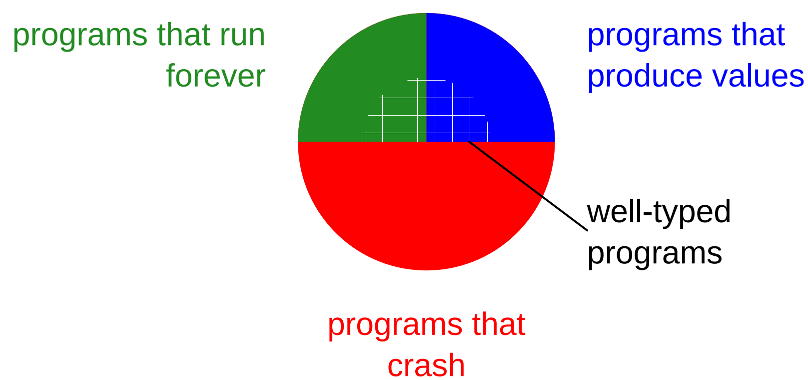
1 / 0 $\longrightarrow\!\!\!\!\rightarrow$ **divide by zero**

The Universe of Programs



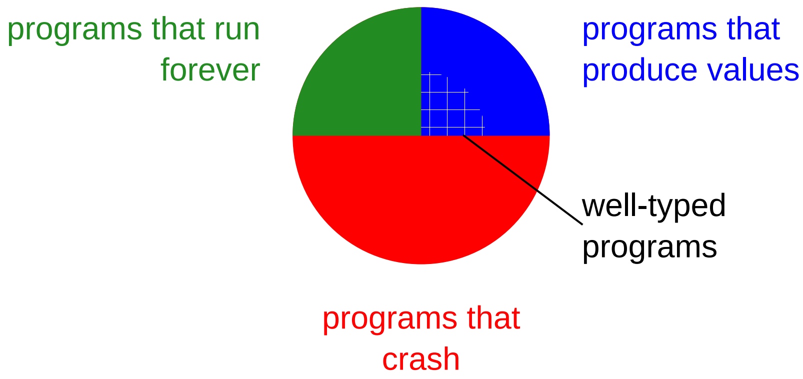
- Still, a type checker *always* rules out a certain class of errors
  - Division by 0 is a ***variant error***

The Universe of Programs



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

The Universe of Programs



- In fact, if we get rid of **letrec**, then every well-typed program terminates with a value!

Intution for Termination

Recall that to get rid of **letrec**

```
letrec int sum = proc(int x)
                   if zero?(x)
                    then 0
                    else +(x,(sum -(x, 1)))
 in (sum 10)
```

we can use self-application:

```
let sum = proc(int x, ? sum)
            if zero?(x)
             then 0
             else +(x,((sum sum) -(x, 1)))
 in ((sum sum) 10)
```

Intution for Termination

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

proc($?_1$ **x**)(**x x**)

T_1 T_1

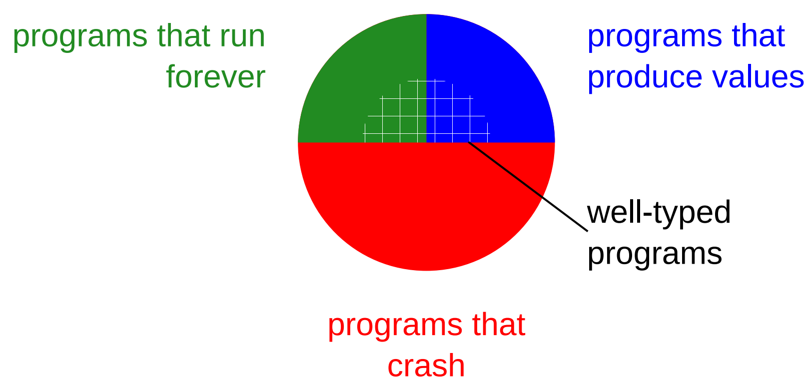
no type: T_1 can't be ($T_1 \rightarrow T_2$)

The only way around this restriction is to restore **letrec** or extend the type language.

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

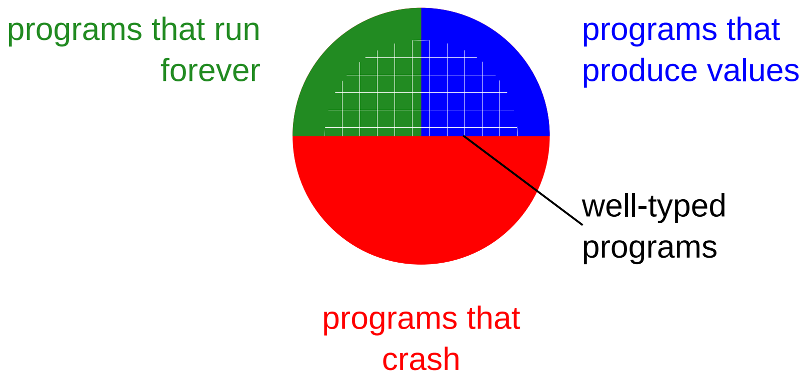
The Universe of Programs

- 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

Polymorphism

proc(?$_1$ **y**)**y**

T$_1$

($\mathbf{T}_1 \rightarrow \mathbf{T}_1$)

let f = **proc**(?$_1$ **y**)**y** : ($\mathbf{T}_1 \rightarrow \mathbf{T}_1$)
in if (**f true**) **then** (**f** 1) **else** (**f** 0)

($\mathbf{T}_1 \rightarrow \mathbf{T}_1$)   ($\mathbf{T}_1 \rightarrow \mathbf{T}_1$)   ($\mathbf{T}_1 \rightarrow \mathbf{T}_1$)

no type: **T**$_1$ can't be both `bool` and `int`

Polymorphism

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

let f = **proc**(?$_1$ **y**)**y** : ($\mathbf{T}_1 \rightarrow \mathbf{T}_1$)
in if (**f true**) **then** (**f** 1) **else** (**f** 0)

($\mathbf{T}_2 \rightarrow \mathbf{T}_2$)   ($\mathbf{T}_3 \rightarrow \mathbf{T}_3$)   ($\mathbf{T}_4 \rightarrow \mathbf{T}_4$)

int

T$_2$ = `bool`   **T**$_3$ = `int`   **T**$_4$ = `int`

- This rule is called ***let-based polymorphism***