Mid-Term 2

- Open book
- Open notes
- Everything through today
    - lexical scope, environments, closures, evaluation, assignment, parameter-passing mechanisms, types
- Example questions on the schedule page

HW9

New construct	*C equivalent*
`ref(x)`	**`&x`**
`setref(E1, E2)`	**`(*E1 = E2, 1)`**

```
let x = 0
 in let y = ref(x)
     in let d = setref(y, 2)
         in x
```

Result: **2**

HW9

```
let x = 0
 in let y = ref(x)
     in let d = setref(y, true)
         in x
```

Result: `true`

But should it be allowed?

HW9

```
let x = 0
 in let y = ref(x)
     in let d = if ...
                 then 1
                 else setref(y, true)
         in +(x, 0)
```

Might crash.

Solution: only allow assignments that do not change a variable’s type

HW9

```
let x = 0 : int
  in let y = ref(x) : (refto int)
     in let d = setref(y, 1)
         in +(x, 0)
```

Ok

HW9

```
let x = 0 : int
  in let y = ref(x) : (refto int)
     in let d = setref(y, true)
         in +(x, 0)
```

Not ok

- First argument of **`setref`** must have type **`(refto T)`**
- Second argument of **`setref`** must have type **`T`**, for the same **`T`**

Back to our regularly scheduled programming...

`: squash`

Type-Checking Expressions

- What is the value of the following expression?

`proc(x)+(x,1)`

- **Answer:** Yet another trick question; it's not an expression in our typed language, because the argument type is missing

- But, clearly, the answer *should* be **`(int -> int)`**

Type Inference

- ***Type inference*** is the process of inserting type annotations where the programmer omits them.

- We'll use explicit question marks, to make it clear where types are being omitted.

$$\texttt{proc (?}_1\texttt{ x)+(x,1)}$$

Type Inference

proc(?$_1$ x)+(x, 1)

T_1 int

int $T_1 = \text{int}$

int -> int

proc(?$_1$ x)if true then 1 else x

bool int T_1

int -> int $T_1 = \text{int}$

proc(?$_1$ x)if x then 1 else x

T_1 int T_1

no type: T_1 can't be both **bool** and **int**

Type Inference

proc(?$_1$ y)y

T_1

T_1 -> T_1

(proc(?$_1$ y)y proc(?$_2$ x)+(x, 1))

T_1 -> T_1 int -> int

int -> int

T_1 = int -> int

proc(?$_1$ y)(y 7)

T_1 int

T_2 T_1 = int -> T_2

(int -> T_2) -> T_2

Type Inference

$\texttt{proc(?}_1\texttt{ x)(x x)}$

T_1 T_1

no type: T_1 can't be T_1 `->` ...

- T_1 can't be `int`
- T_1 can't be `bool`
- Suppose T_1 is T_2 `->` T_3
  - T_2 must be T_1
  - So we won't get anywhere!

Implementation

- Extend **`type`** datatype with **`tvar-type`** variant

```
(define-datatype type type?
  ...
  (tvar-type
    (serial-number integer?)
    (container vector?)))
```

- Create a new type variable record for each **`?`**
  - Initial container value is ‘‘don’t know’’, **`’()`**
- Create a new type variable record for each application
- Change **`check-equal-type!`** to read and set type variable containers

The Universe of Programs

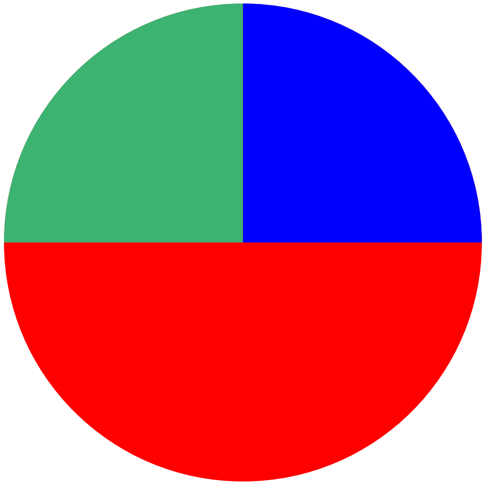
- The goal of type-checking is to rule out bad programs

```
+(1, true)
```

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

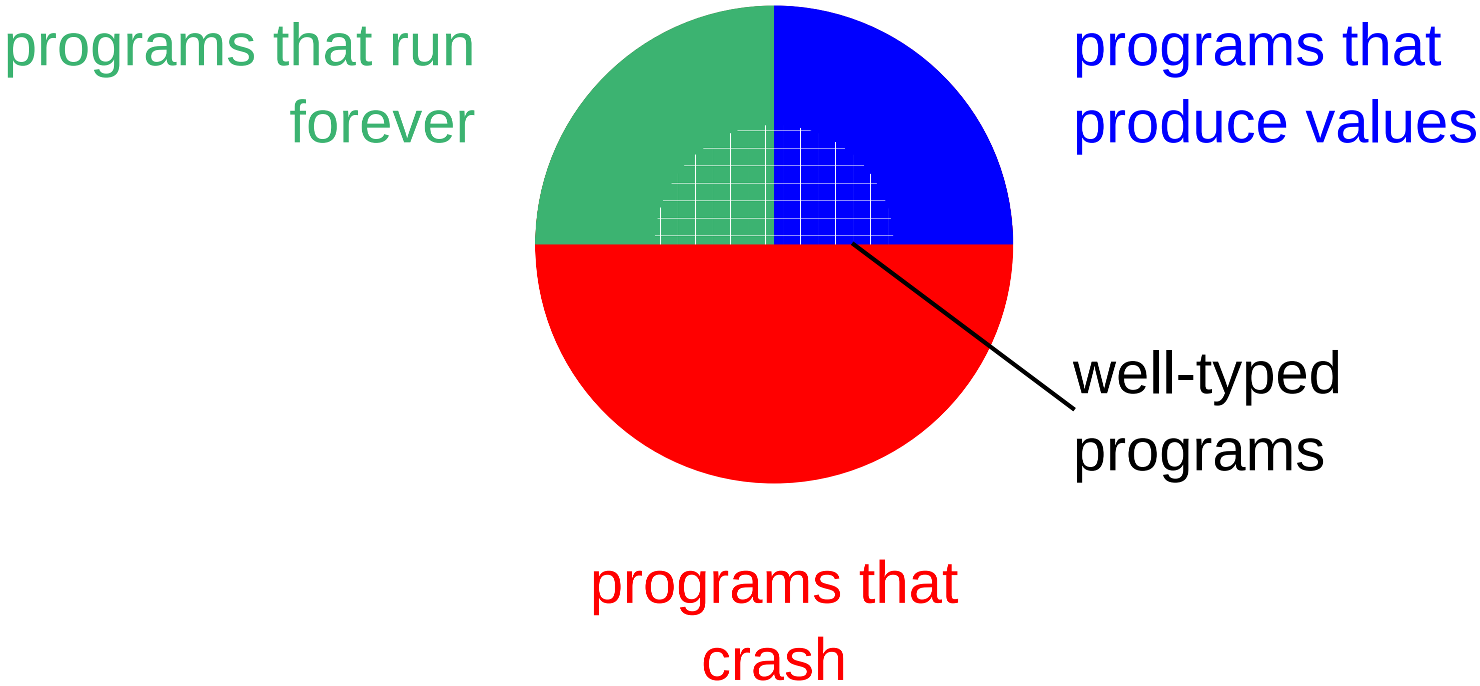
```
+(1, if true then 1 else false)
```

The Universe of Programs



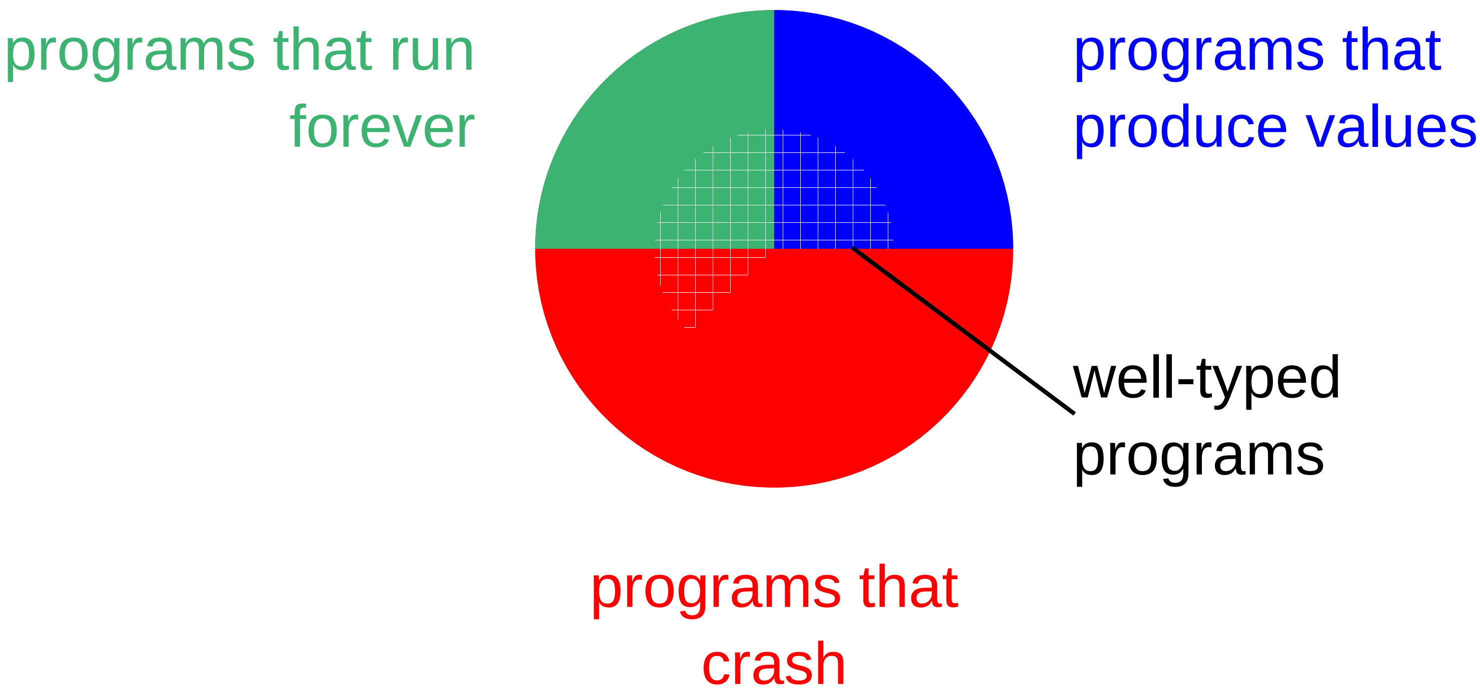
- Every program falls into one of three categories

The Universe of Programs



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

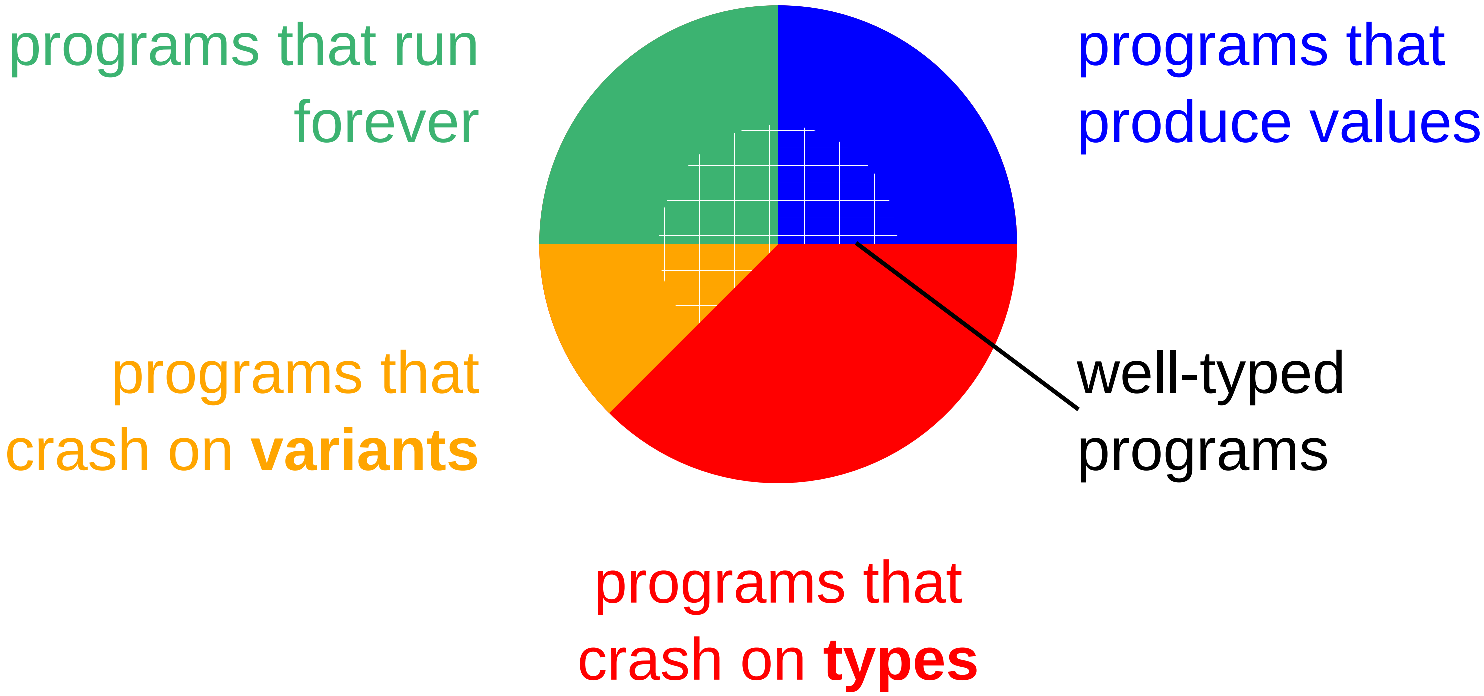
The Universe of Programs



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

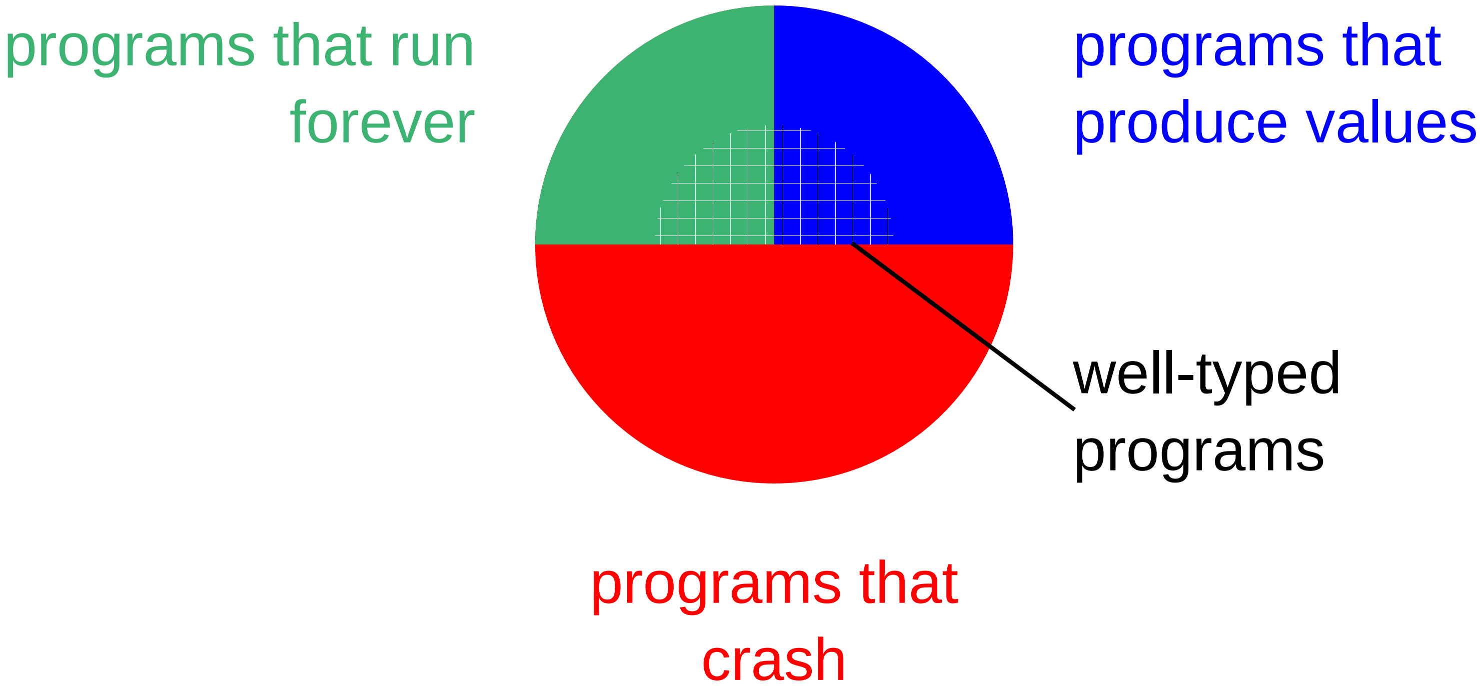
$$\texttt{1 / 0} \Rightarrow \texttt{divide by zero}$$

The Universe of Programs



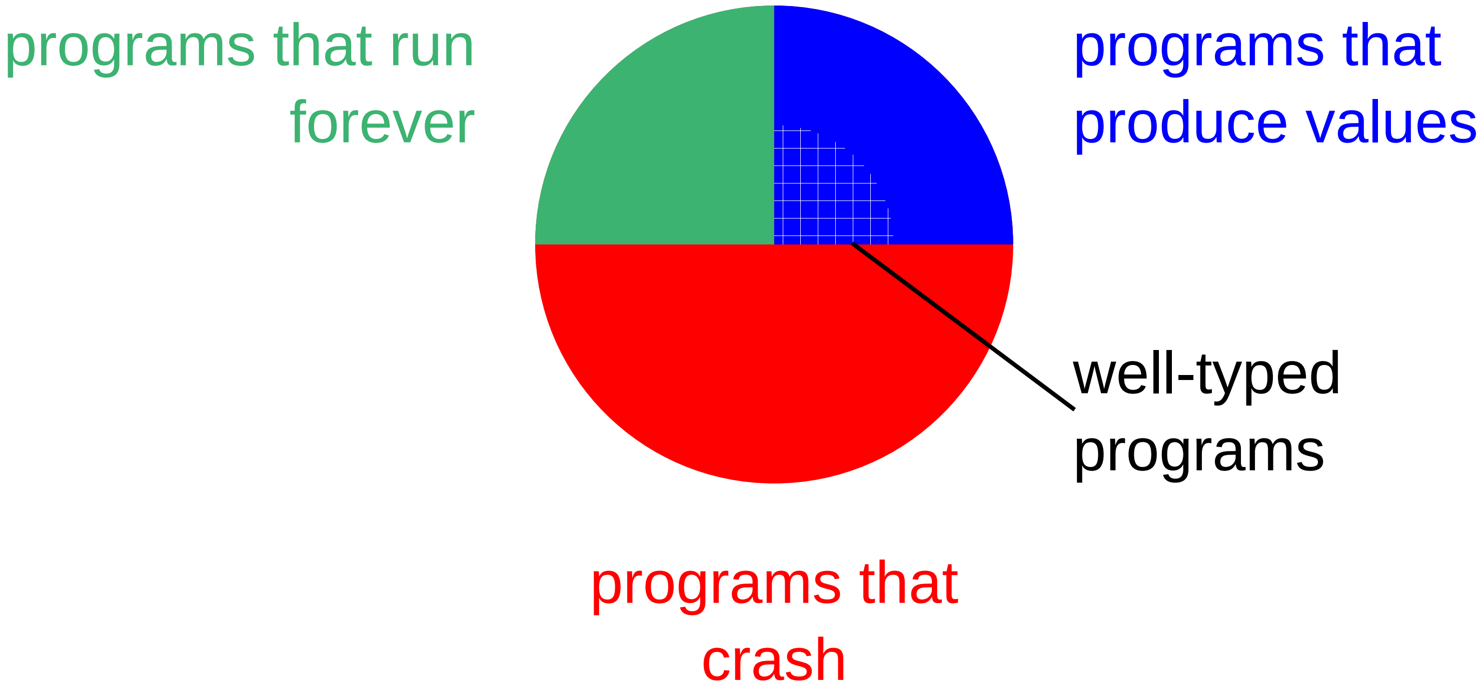
- Still, a type checker *always* rules out a certain class of errors
    - Division by 0 is a ***variant error***

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:

$\texttt{proc(?}_{1}\texttt{ x)(x x)}$

T_1 T_1

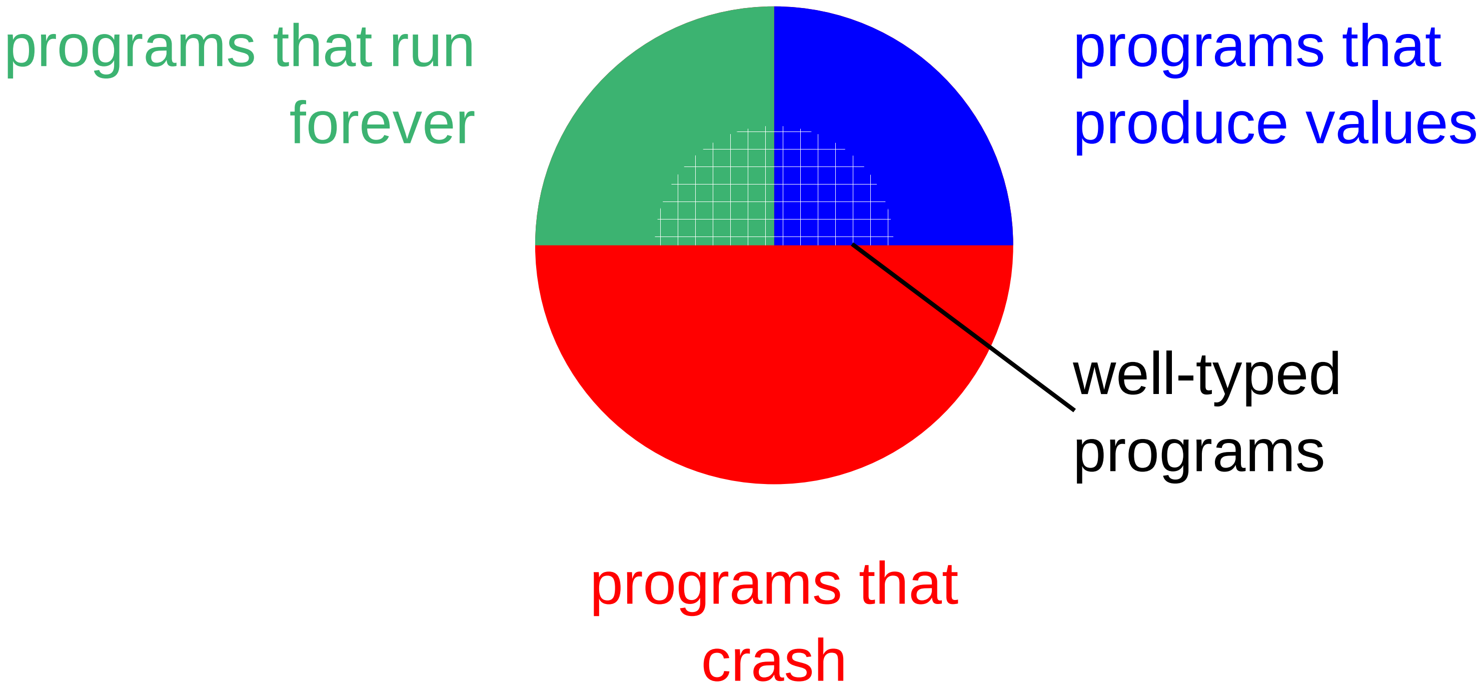
no type: $\mathbf{T_1}$ can't be $\mathbf{T_1}$ **`->`** ...

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.)

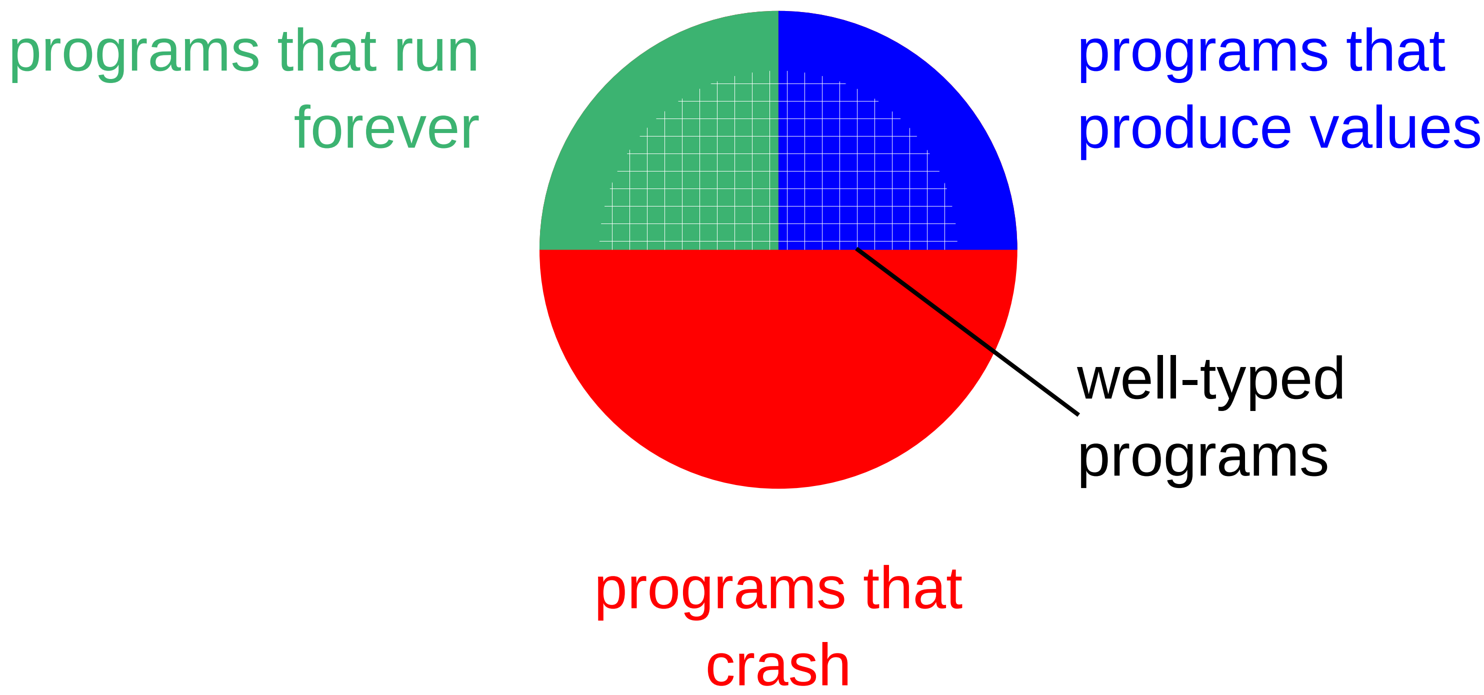
The Universe of Programs

- 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

T_1 -> T_1

let f = prog($?_1$ y)y : T_1 -> T_1
 in if (f true) then (f 1) else (f 0)

T_1 -> T_1 T_1 -> T_1 T_1 -> 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

$$\texttt{let f = prog(?}_1\texttt{ y)y} : T_1 \texttt{ -> } T_1$$

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

$$T_2 \texttt{ -> } T_2 \qquad T_3 \texttt{ -> } T_3 \qquad T_4 \texttt{ -> } T_4$$

$$\texttt{int}$$

$$T_2 = \texttt{bool} \quad T_3 = \texttt{int} \quad T_4 = \texttt{int}$$

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