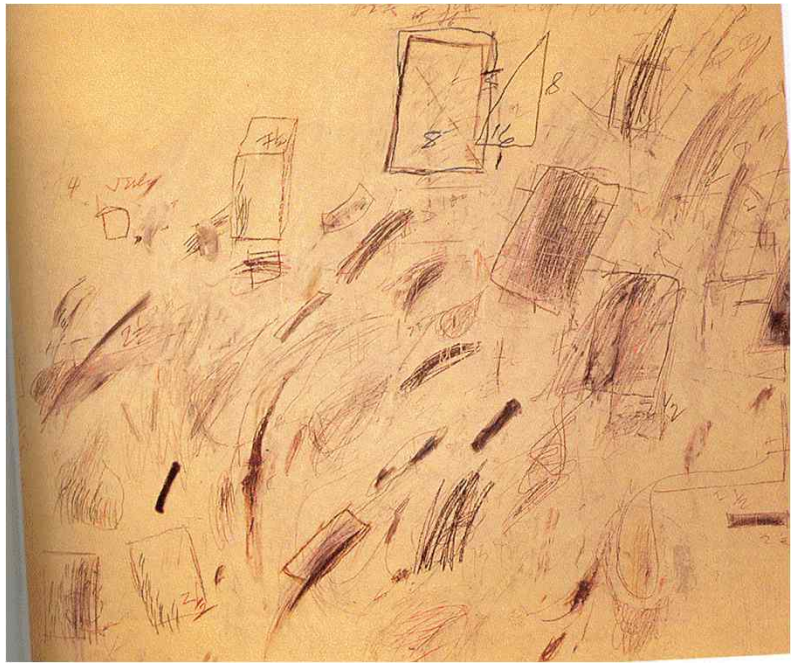
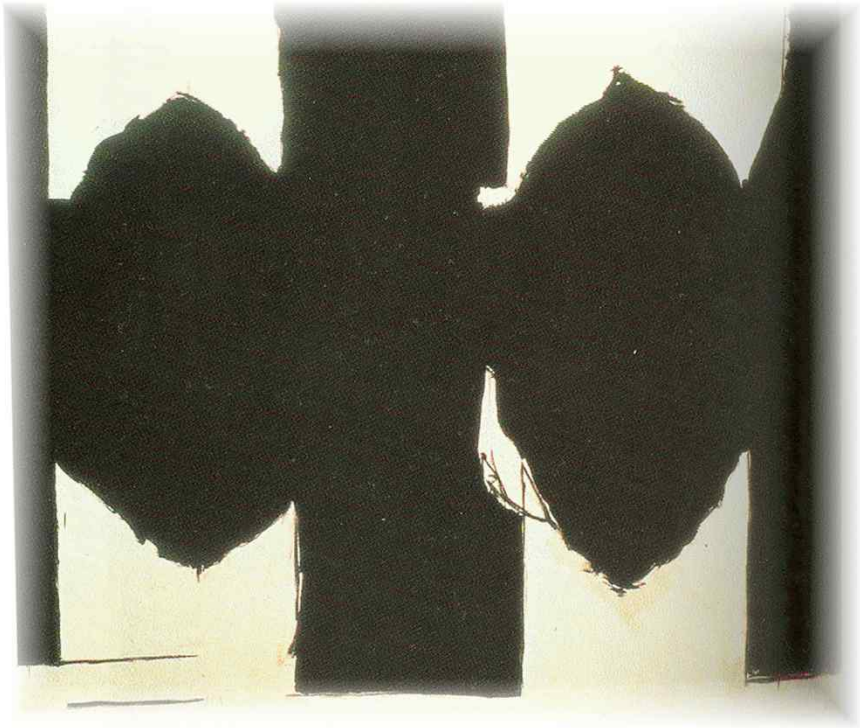
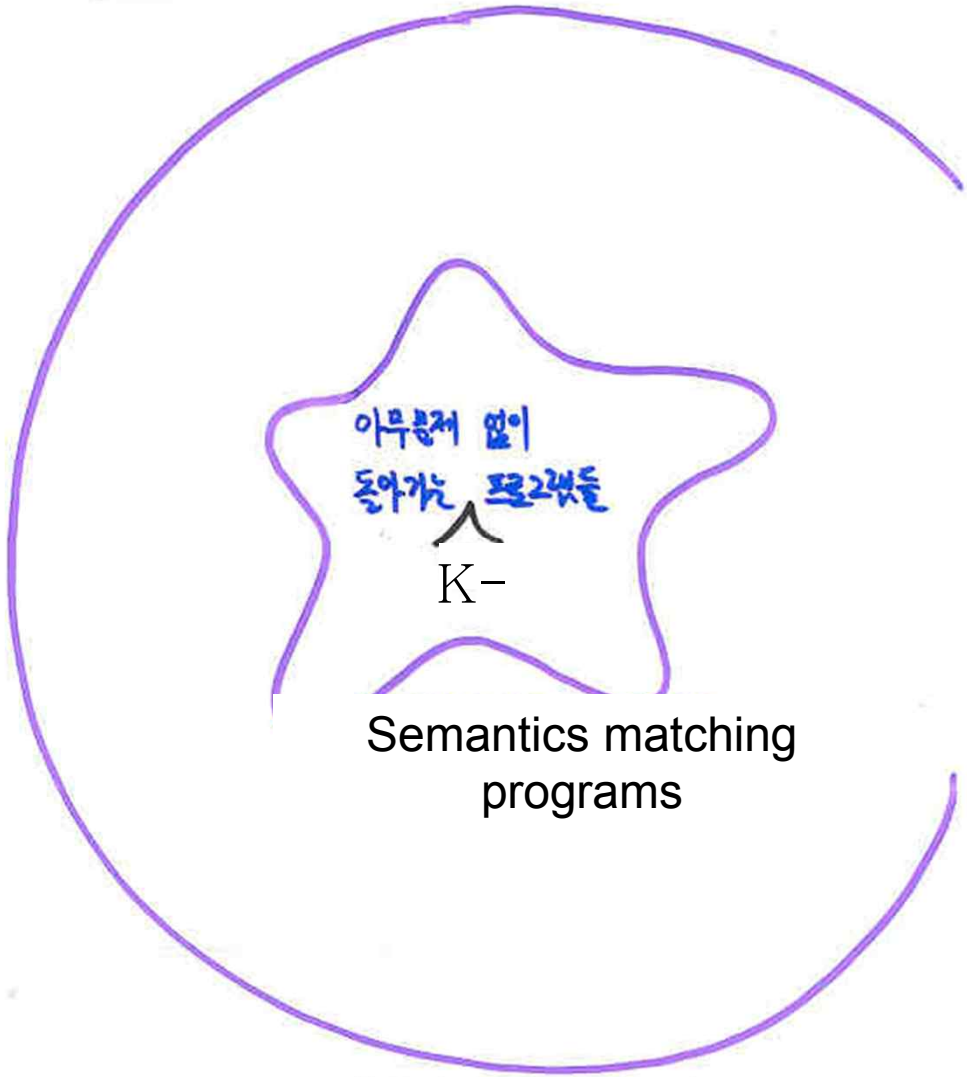
Meaningless programs



- let x := 1
  in call x(2)
- let procedure f(x) = ...
  in f + 3
- let x := 1
  in let y := true
  in x + y
- while 1 do E
- if {name := 1} then 2
- for x := 1 to false do E
- not (E + E)
- if (read x) then 1
- let x := {id := 83031147, age := 19}
  in x.id.age
- let procedure f(x) = ... in call f(f)

Want to run meaningful program!



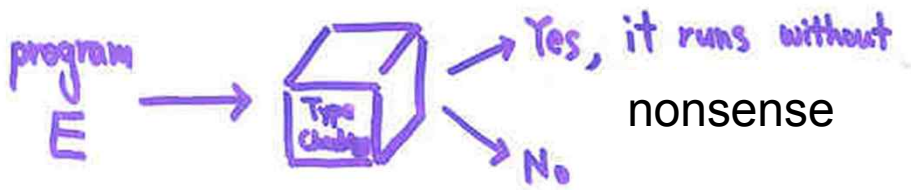


K- Syntax matching programs

Type Checking

* A way to determine wheather a given program has semantics, thus it is trouble-free program or not.

Especially, if the way is sound..



* static type checking : before execution

Before running program

Running program

* dynamic type checking : during execution

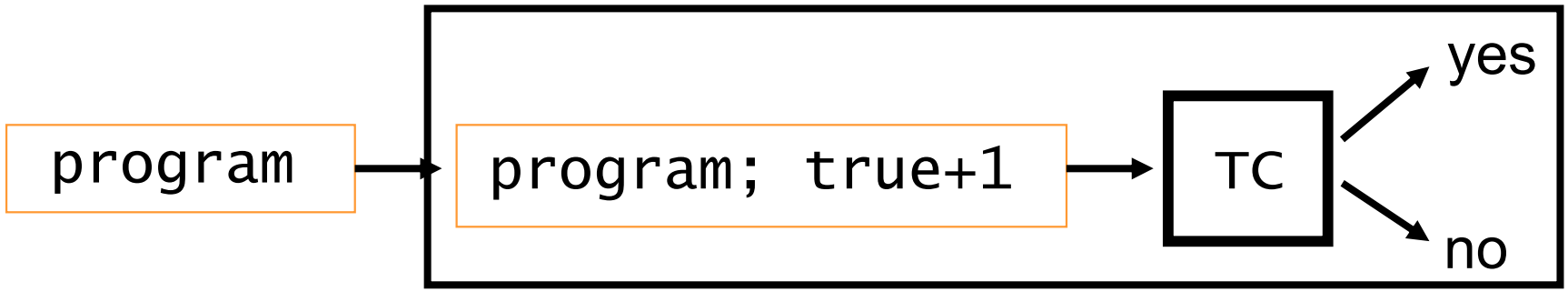
Sound and complete type-checking
is usually impossible.

Only sound type-checking is an alternative.

And we try to make it as complete as possible.

Why is sound & complete type-checking impossible in general?

If such type-checking procedure TC exists, then we can solve the Halting problem as follows:



Types

Form, Look, Shape

E : int

E : bool

E : {x: int, y: bool}

f : int → {age: int, id: int}

E : unit

Type Hierarchies

Logics	Computation
syntax definition	parser
semantic definition	interpreter
type checking rules	type checking procedure

Develop a skills to deal with
two different world.

$$\Gamma, \varphi \vdash \varphi$$

$$\frac{\Gamma \vdash \varphi \quad \Gamma \vdash \psi}{\Gamma \vdash \varphi \wedge \psi} \qquad \frac{\Gamma \vdash \varphi \wedge \psi}{\Gamma \vdash \varphi}$$

$$\frac{\Gamma \vdash \varphi}{\Gamma \vdash \varphi \vee \psi}$$

$$\frac{\Gamma, \varphi \vdash \rho \quad \Gamma, \psi \vdash \rho \quad \Gamma \vdash \varphi \vee \psi}{\Gamma \vdash \rho}$$

$$\frac{\Gamma, \varphi \vdash \psi}{\Gamma \vdash \varphi \to \psi} \qquad \frac{\Gamma \vdash \varphi \to \psi \quad \Gamma \vdash \varphi}{\Gamma \vdash \psi}$$

$$\dfrac{\dfrac{\varphi, \psi \vdash \varphi}{\varphi \vdash \psi \to \varphi}}{\vdash \varphi \to (\psi \to \varphi)}$$

$$E \rightarrow n \mid E + E \mid E \times E$$

E : even | E : odd

$$\frac{n \ mod \ 2 = 0}{n : \texttt{even}} \qquad \frac{n \ mod \ 2 = 1}{n : \texttt{odd}}$$

$$\frac{E_1 : \texttt{even} \quad E_2 : \texttt{even}}{E_1 + E_2 : \texttt{even}}$$

$$\frac{E_1 : \texttt{odd} \quad E_2 : \texttt{odd}}{E_1 + E_2 : \texttt{even}}$$

$$
\begin{array}{rcl}
E & \rightarrow & n \mid E + E \mid E \times E \\
& \mid & x \mid \texttt{let}\ x = E\ \texttt{in}\ E
\end{array}
$$

$x + 1 : \texttt{even?}$ $x + 1 : \texttt{odd?}$

All depends on whether x is odd or even

All depends on assumption about x

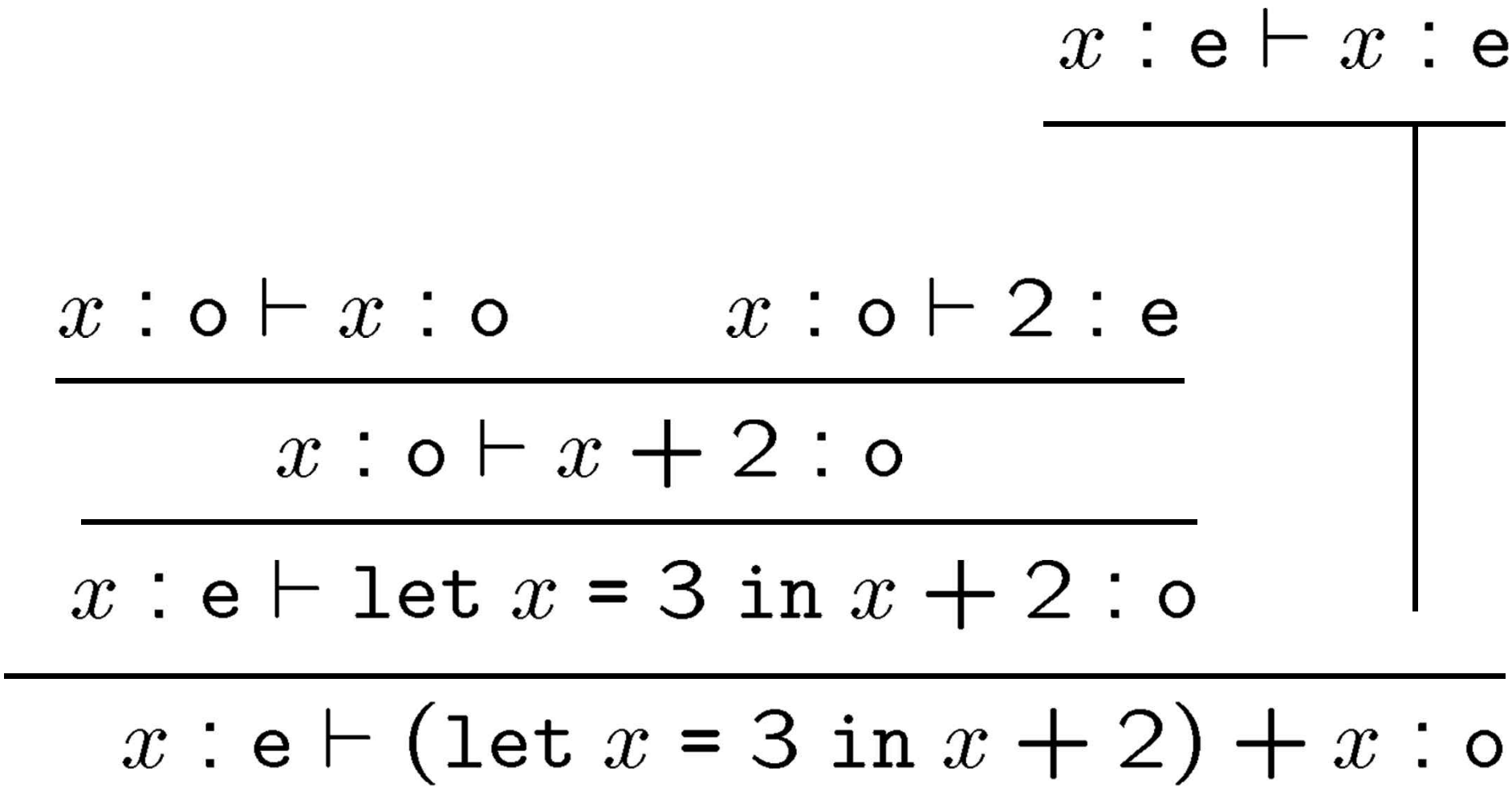
$$\Gamma \vdash E : t$$

$$\frac{n \ mod \ 2 = 0}{\Gamma \vdash n : \texttt{even}}$$

$$\frac{\Gamma \vdash E_1 : \texttt{even} \quad \Gamma \vdash E_2 : \texttt{even}}{\Gamma \vdash E_1 + E_2 : \texttt{even}}$$

$$\frac{\Gamma(x) = t}{\Gamma \vdash x : t}$$

$$\frac{\Gamma \vdash E_1 : t_1 \quad \Gamma[t_1/x] \vdash E_2 : t}{\Gamma \vdash \texttt{let } x\texttt{= } E_1 \texttt{ in } E_2 : t}$$


$x : \mathtt{e} \vdash x : \mathtt{e}$
$x : \mathtt{o} \vdash x : \mathtt{o}$
$x : \mathtt{o} \vdash 2 : \mathtt{e}$
$x : \mathtt{o} \vdash x + 2 : \mathtt{o}$
$x : \mathtt{e} \vdash \mathtt{let}\ x = 3\ \mathtt{in}\ x + 2 : \mathtt{o}$
$x : \mathtt{e} \vdash (\mathtt{let}\ x = 3\ \mathtt{in}\ x + 2) + x : \mathtt{o}$

$$\Gamma \vdash E : \tau$$

K-의 Types

$\tau \rightarrow$ int
| bool
| unit
| $\{x_1 \mapsto \tau, \cdots, x_n \mapsto \tau\}$
| $\tau \rightarrow \tau$
| τ var

} monomorphic types

Let's find the below expression's types

- let x := 1
  in x

- let procedure f(x) = x := 1 ; x
  in let y := 0
  in call f(y) + y

- let x := {name := 1, age := 2}
  in x.name := x.age

- if E then 1 else 2

- if E then 1
  else true

$$\Gamma \vdash n : \texttt{int} \qquad \Gamma \vdash true : \texttt{bool}$$

$$\Gamma \vdash () : \texttt{unit}$$

$$\frac{\Gamma \vdash E_1 : \texttt{int} \quad \Gamma \vdash E_2 : \texttt{int}}{\Gamma \vdash E_1 + E_2 : \texttt{int}}$$

$$\frac{\Gamma \vdash E_1 : \texttt{bool} \quad \Gamma \vdash E_2 : \tau \quad \Gamma \vdash E_3 : \tau}{\Gamma \vdash \texttt{if } E_1 \texttt{ then } E_2 \texttt{ else } E_3 : \tau}$$

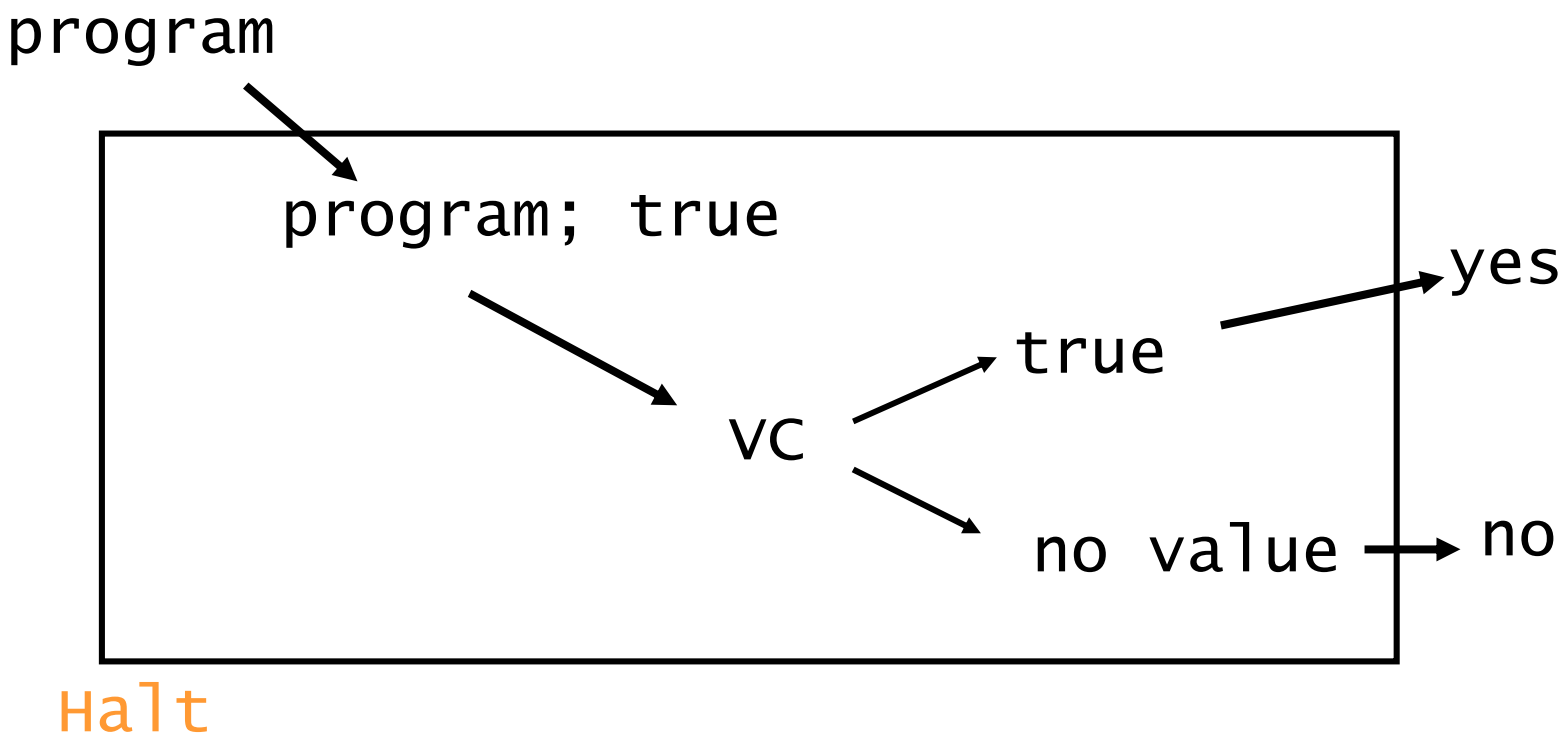
Why two branches should have the same type?

(if E then 1 else true) + 1

We might know the value of E in advance
Always true, Above program is OK!
Or, always false, no OK!
Or, true/false both possible, no OK!
Or, no value, OK!

There are no way to know the value of E correctly without running.

I can solve the Halting Problem
if I have the way(VC):



$$\frac{\Gamma \vdash E_1 : \texttt{bool} \qquad \Gamma \vdash E_2 : \texttt{unit}}{\Gamma \vdash \texttt{while}\ E_1\ E_2 : \texttt{unit}}$$

Type Checking K- Programs

$\Gamma \vdash E : \tau$

$\Gamma \vdash n : \text{int}$ $\quad$ $\Gamma \vdash \text{true} : \text{bool}$ $\quad$ $\Gamma \vdash () : \text{unit}$

$$\frac{\Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int}}{\Gamma \vdash E_1 + E_2 : \text{int}}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma \vdash E_2 : \tau_2}{\Gamma \vdash E_1 ; E_2 : \tau_2}$$

$$\frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : \tau \quad \Gamma \vdash E_3 : \tau}{\Gamma \vdash \text{if } E_1 \text{ then } E_2 \text{ else } E_3 : \tau}$$

$$\frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : \text{unit}}{\Gamma \vdash \text{if } E_1 \text{ then } E_2 : \text{unit}}$$

$$\frac{\Gamma \vdash E_1 : \text{bool} \quad \Gamma \vdash E_2 : \text{unit}}{\Gamma \vdash \text{while } E_1 \text{ do } E_2 : \text{unit}}$$

$$\frac{\Gamma(x) = \text{int var} \quad \Gamma \vdash E_1 : \text{int} \quad \Gamma \vdash E_2 : \text{int} \quad \Gamma \vdash E_3 : \text{unit}}{\Gamma \vdash \text{for } x := E_1 \text{ to } E_2 \text{ do } E_3 : \text{unit}}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \Gamma[\tau_1 \text{ var}/x] \vdash E_2 : \tau_2}{\Gamma \vdash \text{let } x := E_1 \text{ in } E_2 : \tau_2}$$

$$\frac{\Gamma \vdash E_1 : \tau \quad \Gamma(x) = \tau \text{ var}}{\Gamma \vdash x := E_1 : \text{unit}}$$

$$\frac{\Gamma(x) = \tau \text{ var}}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma[\tau \text{ var}/y][\tau \to \tau_1/x] \vdash E_1 : \tau_1 \quad \Gamma[\tau \to \tau_1/x] \vdash E_2 : \tau_2}{\Gamma \vdash \text{let procedure } x(y) = E_1 \text{ in } E_2 : \tau_2}$$

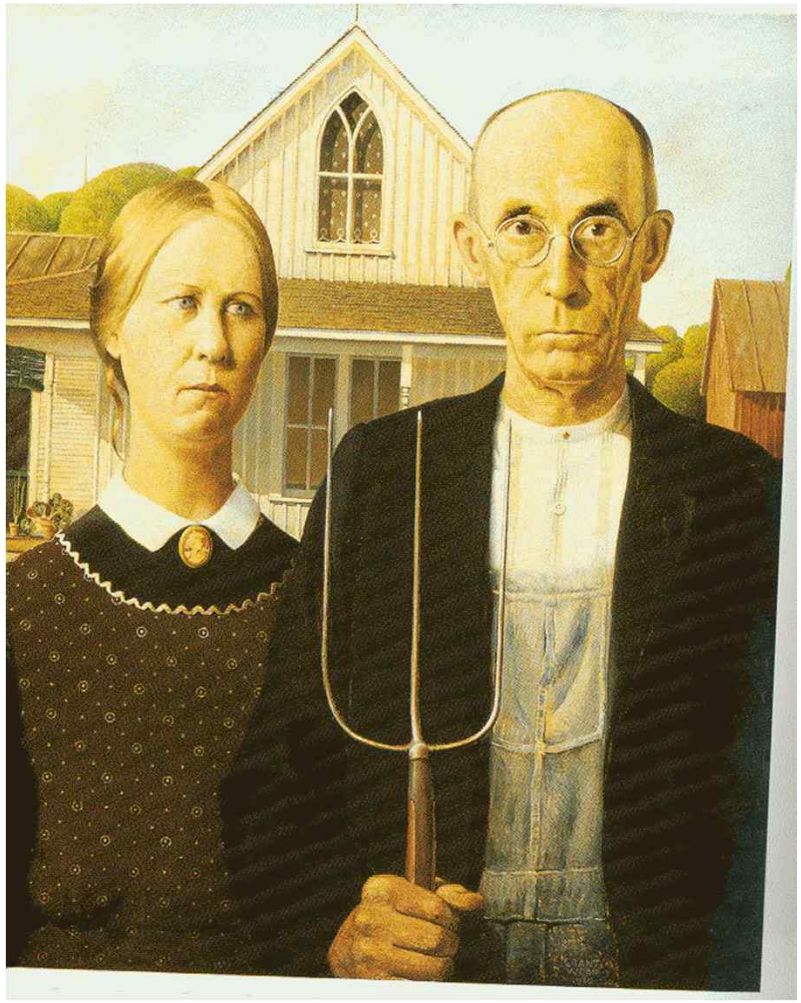
$$\frac{\Gamma \vdash E : \tau_1 \quad \Gamma(x) = \tau_1 \to \tau_2}{\Gamma \vdash \text{call } x(E) : \tau_2}$$

$$\frac{\Gamma \vdash E_1 : \tau_1 \quad \vdots \quad \Gamma \vdash E_n : \tau_n}{\Gamma \vdash \{x_1 := E_1; \cdots; x_n := E_n\} : \{x_1 \mapsto \tau_1\ \mathrm{var}, \ \vdots \ x_n \mapsto \tau_n\ \mathrm{var}\}}$$

$$\frac{\Gamma \vdash E : \tau \qquad \tau(x) = \tau_1\ \mathrm{var}}{\Gamma \vdash E.x : \tau_1}$$

$$\frac{\Gamma \vdash E_1 : \tau \qquad \tau(x) = \tau'\ \mathrm{var} \qquad \Gamma \vdash E_2 : \tau'}{\Gamma \vdash E_1.x := E_2 : \mathrm{unit}}$$

$$\frac{\Gamma \vdash E_1 : \tau \qquad \Gamma \vdash E_2 : \tau}{\Gamma \vdash E_1 = E_2 : \mathrm{bool}}$$



어서보자 Observations

* Memory address preserves the type when it is allocated

```
let x := 1                    "reject!"
in x := true
```

* Just one type is possible

```
let procedure f(x) = x        "reject!"
in f(1); f(true)
```

```
let procedure f(x) = x.age := 19
in f({age:=1, id:=001});
   f({age:=2, height:=170})

                              "reject!"
```

* Have to inference the type of a function..

$$\frac{\Gamma[\tau_{var}/y][\tau \to \tau_1/x] \vdash E_1 : \tau_1 \qquad \Gamma[\tau \to \tau_1/x] \vdash E_2 : \tau}{\Gamma \vdash \text{let procedure } x(y) = E_1 \text{ in } E_2 : \tau}$$

* Isn't the record type too strict?

```
let
    node := { x:=0, next :={}}
in
    node.next := { x:=1, next :={}}
```

"reject!"

Let's lean on Programmer.

K-: Type Declarations & Annotations

* helps/simplifies type checking
* as machine-checkable comments

Program $P \rightarrow T^* E$

Type $T \rightarrow$ type $x = \{t_1 x_1, \cdots, t_n x_n\}$

$t \rightarrow x$
| int
| bool
| unit

Expression $E \rightarrow \cdots$
| let procedure $f(t\,x) : t = E$ in E
| let $t\,x := E$ in E

e.g.)

```
type bbs = { int name, bool zap}
let bbs x := { name := 0, zap := true}
in if x.zap then 1
           else x.name
```

e.g.)

```
type intlist = { int x, intlist next}
let intlist l := { x:=0, next := {}}
in
    l.next := { x:=1, next := {}} ;
    l.next.next := {x:=2, next :={}}
```

e.g.)

```
type intree = { intree l, int x, intree r}
let procedure shake (intree *) : intree
  = if * = {} then *
    else let intree *' := {}
         in *' := call shake (*.l);
            *.l := call shake(*.r);
            *.r := *'
in call shake({ l:={}, x:=1, r := { l:={}, x:=2, r:={}}
              })
```

$$
\begin{array}{lcll}
\Gamma & \in & VarEnv & = & Var \to Type \\
\Delta & \in & TypeEnv & = & Tname \to Type \\
\tau & \in & Type\ \tau & \to & \texttt{unit} \mid \texttt{bool} \mid \texttt{int} \\
 & & & \mid & \tau\texttt{->}\tau \mid \tau\ \texttt{var} \\
 & & & \mid & x
\end{array}
$$

$$\boxed{\Delta \vdash T^* : \Delta'} \qquad \boxed{\Gamma, \Delta \vdash E : \tau}$$

$$\frac{\emptyset \vdash T^* : \Delta \qquad \emptyset, \Delta \vdash E : \tau}{\emptyset, \emptyset \vdash T^*\ E : \tau}$$

$$\frac{\Delta \vdash T_1 : \Delta_1 \qquad \Delta_1 \vdash T_2 : \Delta_2}{\Delta \vdash T_1\ T_2 : \Delta_2}$$

$$\frac{x_1 \neq x_2}{\begin{array}{c}\Delta \vdash \texttt{type x =}\{t_1\ x_1,\ t_2\ x_2\} : \\ \Delta[\{x_1 \mapsto t_1\ \texttt{var}, x_2 \mapsto t_2\ \texttt{var}\}/\texttt{x}]\end{array}}$$

$$\frac{\Gamma, \Delta \vdash E : \tau \quad \Delta(y) = \{x \mapsto \tau\ \texttt{var}\}}{\Gamma, \Delta \vdash \texttt{\{x:=E\}} : y}$$

$$\frac{}{\Gamma, \Delta \vdash \texttt{\{\}} : x}$$

$$\frac{\Gamma[t \to t'/\texttt{f}][t\ \texttt{var}/\texttt{x}], \Delta \vdash E : t'}{\Gamma, \Delta \vdash \ \texttt{proc f(}t\ \texttt{x)}\texttt{:}t'\ \texttt{=}\ E : \mathsf{ok}}$$

어디보자 II
Observations

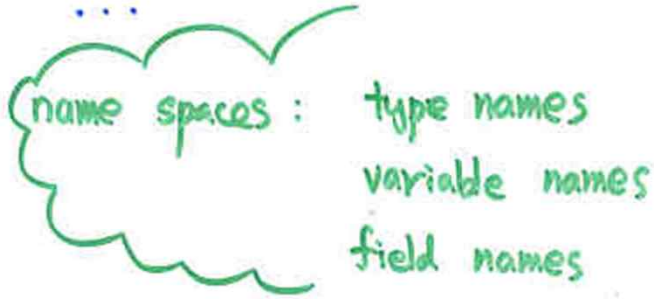
* let x := { a:=1, b:=2} in x end

* type 리스트 = {int x, 리스트 next}
  type 트리스 = {int x, 트리스 prev}
  let
    ... { x:=1, next:={}}
        { x:=1, prev:={}}
  ...

* type t1 = {int x, bool y}
  type t2 = {bool y, int x}
  let ... { x:=1, y:=true}
  ...

* type 짱근 = { int x, 멍근 y}
  type 멍근 = { bool x, 짱근 z}
  let 짱근 x:={ x:=1, y:={}}
      멍근 x':={ x:=true, z:={}}
  in
    x.y := x';
    x'.z := x
  end

* type x = {int x, bool y}
  type y = {int id, x employ}
  let
    x x := {x := 1, y := true}
    y y := {id := 2, employ := x}
  in
  ...



* every record type has to be named ?

What would you do to remove this restriction ?

- can you change the language's syntax & type system for it ?
- go ahead

We've been with flow,
We defined plausible
type checking rules,
but,

It seems that
somewhere is a third
rate.
Irresponsible.
Descendants laugh.
Uncivilized.

Our hope

Theorem [Type Safety]

Let E be a program.
If E is type-checked OK, then it does not go wrong.

Theorem [Type Safety]

Let E be a program.
If {} |- E: t then E does not go wrong.

Theorem [Type Safety]

Let E be a program.
If {} |- E : t then 돌렷({},{},E) runs OK.

* `type 식은 = { int x, 식은 next }`

```
let
  식은 node = { x := 1, next := {} }
in
  node.next.x
end
```

type-checked ! but cannot run :(

What would you do to make the type system safe ?

- what did we miss ?