# 타입으로 안전하게 프로그래밍하기

이재호

2024년 1월 13일

SNULife 개발팀

# 목차

# 타입 시스템이란?

# “자유로운” 언어의 숙명

ESLint

Team Blog **Docs** Store

Version

v8.56.0

Search

USE ESLINT IN YOUR PROJECT

Getting Started

Core Concepts

Configure ESLint

Configuration Files (New)

Configuration Files

Configure Language Options

Configure Rules

Configure Plugins

## no-unsafe-optional-chaining

Disallow use of optional chaining in contexts where the `undefined` value is not allowed

✅ The `"extends": "eslint:recommended"` property in a configuration file enables this rule

**Table of Contents**

- Rule Details
- Options
  - disallowArithmeticOperators
- Version
- Resources

The optional chaining (`?.`) expression can short-circuit with a return value of `undefined`. Therefore, treating an evaluated optional chaining expression as a function, object, number, etc., can cause

# “자유로운” 언어의 숙명

*This rule aims to detect some cases where the use of optional chaining doesn’t prevent runtime errors. In particular, it flags optional chaining expressions in positions where short-circuiting to* `undefined` *causes throwing a TypeError afterward.*

```
/*eslint no-unsafe-optional-chaining: "error"*/
(obj?.foo)();
(obj?.foo).bar;
(foo?.()).bar;
(foo?.()).bar();
```

# “자유로운” 언어의 숙명

## unsupported-binary-operation / E1131

**Message emitted:**

`%s`

**Description:**

*Emitted when a binary arithmetic operation between two operands is not supported.*

**Problematic code:**

```
drink = "water" | None  # [unsupported-binary-operation]
result = [] | None  # [unsupported-binary-operation]
```

**Correct code:**

```
masked = 0b111111 & 0b001100
result = 0xAEFF | 0x0B99
```

# 동적 타입과 정적 타입

동적 타입 시스템은 사실 “타입 시스템”이 아닌 실행 중 타입 태그 검사

**동적 타입 언어** 파이썬 *Python*, 자바스크립트 *JavaScript*, 스킴 *Scheme*, 라켓 *Racket*, 루아 *Lua*, 루비 *Ruby*, 펄 *Perl*, 라쿠 *Raku*, …

**정적 타입 언어** 스위프트 *Swift*, 코틀린 *Kotlin*, 러스트 *Rust*, 오캐멀 *OCaml*, 하스켈 *Haskell*, 리스크립트 *ReScript*, 엘 *Elm*, C, C++, 자바 *Java*, …

# 동적 타입과 정적 타입

동적 타입 시스템은 사실 “타입 시스템”이 아닌 실행 중 타입 태그 검사

**동적 타입 언어** 파이썬 *Python*, 자바스크립트 *JavaScript*, 스킴 *Scheme*, 라켓 *Racket*, 루아 *Lua*, 루비 *Ruby*, 펄 *Perl*, 라쿠 *Raku*, …

**정적 타입 언어** 스위프트 *Swift*, 코틀린 *Kotlin*, 러스트 *Rust*, 오캐멀 *OCaml*, 하스켈 *Haskell*, 리스크립트 *ReScript*, 엘름 *Elm*, C, C++, 자바 *Java*, …

```
Python 3.11.4 (main, Jul  4 2023, 19:38:47) [Clang 14.0.3 (clang-1403.0.22.14.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> def poison():
...     "hello" + 42
...
>>> poison()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in poison
TypeError: can only concatenate str (not "int") to str
```

# 동적 타입과 정적 타입

## 동적 타입 시스템은 사실 “타입 시스템”이 아닌 실행 중 타입 태그 검사

**동적 타입 언어** 파이썬 *Python*, 자바스크립트 *JavaScript*, 스킴 *Scheme*, 라켓 *Racket*, 루아 *Lua*, 루비 *Ruby*, 펄 *Perl*, 라쿠 *Raku*, …

**정적 타입 언어** 스위프트 *Swift*, 코틀린 *Kotlin*, 러스트 *Rust*, 오캐멀 *OCaml*, 하스켈 *Haskell*, 리스크립트 *ReScript*, 엘름 *Elm*, C, C++, 자바 *Java*, …

```
Welcome to Apple Swift version 5.9.2 (swiftlang-5.9.2.2.56 clang-1500.1.0.2.5).
Type :help for assistance.
  1> func poison() {
  2.     "hello" + 42
  3. }
error: repl.swift:2:13: error: binary operator '+' cannot be applied to operands of type 'String'
↪  and 'Int'
    "hello" + 42
    ~~~~~~~ ^ ~~

repl.swift:2:13: note: overloads for '+' exist with these partially matching parameter lists: (Int,
↪  Int), (String, String)
    "hello" + 42
            ^
```

# 동적 타입과 정적 타입

동적 타입 시스템은 사실 “타입 시스템”이 아닌 실행 중 타입 태그 검사

**동적 타입 언어** 파이썬 *Python*, 자바스크립트 *JavaScript*, 스킴 *Scheme*, 라켓 *Racket*, 루아 *Lua*, 루비 *Ruby*, 펄 *Perl*, 라쿠 *Raku*, …

**정적 타입 언어** 스위프트 *Swift*, 코틀린 *Kotlin*, 러스트 *Rust*, 오캐멀 *OCaml*, 하스켈 *Haskell*, 리스크립트 *ReScript*, 엘 *Elm*, C, C++, 자바 *Java*, …

```
OCaml version 5.1.0
Enter #help;; for help.

# let poison () = "hello" + 42;;
Error: This expression has type string but an expression was expected of type
         int
```

# 동적 타입과 정적 타입

## 동적 타입 시스템은 사실 “타입 시스템”이 아닌 실행 중 타입 태그 검사

**동적 타입 언어** 파이썬 *Python*, 자바스크립트 *JavaScript*, 스킴 *Scheme*, 라켓 *Racket*, 루아 *Lua*, 루비 *Ruby*, 펄 *Perl*, 라쿠 *Raku*, …

**정적 타입 언어** 스위프트 *Swift*, 코틀린 *Kotlin*, 러스트 *Rust*, 오캐멀 *OCaml*, 하스켈 *Haskell*, 리스크립트 *ReScript*, 엘 *Elm*, C, C++, 자바 *Java*, …

```
Welcome to Node.js v20.10.0.
Type ".help" for more information.
> function poison() {
... "hello" + 42
... }
undefined
> poison()
undefined
> "hello" + 42
'hello42'
```

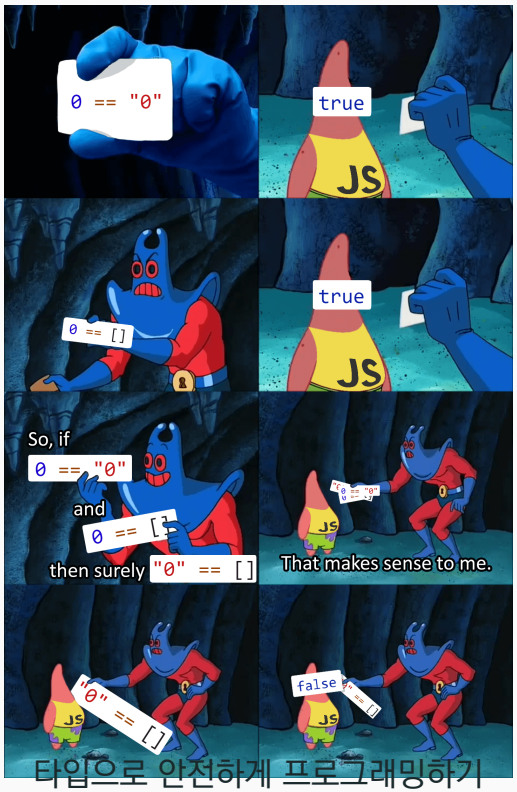
???

# 동적 타입과 정적 타입

## 동적 타입 시스템은 사실 “타입 시스템”이 아닌 실행 중 타입 태그 검사

**동적 타입 언어** 파이썬 *Python*, 자바스크립트 *JavaScript*, 스킴 *Scheme*, 라켓 *Racket*, 루아 *Lua*, 루비 *Ruby*, 펄 *Perl*, 라쿠 *Raku*, …

**정적 타입 언어** 스위프트 *Swift*, 코틀린 *Kotlin*, 러스트 *Rust*, 오캐멀 *OCaml*, 하스켈 *Haskell*, 리스크립트 *ReScript*, 엘름 *Elm*, C, C++, 자바 *Java*, …



# 타입 시스템

타입 시스템은 **실행 전**에 프로그램이 **잘** 실행될 수 있는지를 **겉모습**으로 검사하는 방법

# 타입 시스템

타입 시스템은 **실행 전**에 프로그램이 **잘** 실행될 수 있는지를 **겉모습**으로 검사하는 방법

> **실행 전에** 오류를 안전하게 *sound* 잡아냄

# 타입 시스템

타입 시스템은 **실행 전**에 프로그램이 **잘** 실행될 수 있는지를 **겉모습**으로 검사하는 방법

**실행 전에** 오류를 안전하게 *sound* 잡아냄

**잘** 만족해야 하는 성질을 타입 시스템에 녹여낼 수 있고, 실행 중에 어긋나는 일이 없음

# 타입 시스템

타입 시스템은 **실행 전**에 프로그램이 **잘** 실행될 수 있는지를 **겉모습**으로 검사하는 방법

**실행 전에** 오류를 안전하게 *sound* 잡아냄

**잘** 만족해야 하는 성질을 타입 시스템에 녹여낼 수 있고, 실행 중에 어긋나는 일이 없음

**겉모습** , 즉 문법적으로 검사를 할 수 있음

# 타입 시스템의 장점

# 타입 시스템의 장점

## 안전성 *Soundness*

# 잘 타입된 프로그램은 잘못되지 않는다

"Well-type programs cannot go wrong" — Robin Milner, in *A Theory of Type Polymorphism in Programming* (1978)

(안전한 *Sound*) 타입 시스템을 갖춘 언어에서는 실행 중 타입 에러가 발생하지 않는다!

# 안전한 타입 시스템이란?

안전성은 실행 중에 “잘” 만족해야 하는 성질이 어긋나지 않는 것을 의미

- 그 성질은 언어마다 다름
- 통상적인 타입 시스템은 실행 과정에서 계산된 값이 실제로 타입된 것과 일치하는지를 의미

# 안전한 타입 시스템이란?

안전성은 실행 중에 “잘” 만족해야 하는 성질이 어긋나지 않는 것을 의미

- 그 성질은 언어마다 다름
- 통상적인 타입 시스템은 실행 과정에서 계산된 값이 실제로 타입된 것과 일치하는지를 의미

안전한 타입 시스템이 중요한 이유

- 타입을 통해 원하는 성질을 표현할 수 있다면, 실제로 값이 그 성질을 만족한다고 신뢰할 수 있음

# 안전한 타입 시스템이란?

안전성은 실행 중에 “잘” 만족해야 하는 성질이 어긋나지 않는 것을 의미

- 그 성질은 언어마다 다름
- 통상적인 타입 시스템은 실행 과정에서 계산된 값이 실제로 타입된 것과 일치하는지를 의미

안전한 타입 시스템이 중요한 이유

- 타입을 통해 원하는 성질을 표현할 수 있다면, 실제로 값이 그 성질을 만족한다고 신뢰할 수 있음
  - 리스트가 비어있지 않다는 성질을 **보장**하는 타입?

# 안전한 타입 시스템이란?

안전성은 실행 중에 “잘” 만족해야 하는 성질이 어긋나지 않는 것을 의미

- 그 성질은 언어마다 다름
- 통상적인 타입 시스템은 실행 과정에서 계산된 값이 실제로 타입된 것과 일치하는지를 의미

안전한 타입 시스템이 중요한 이유

- 타입을 통해 원하는 성질을 표현할 수 있다면, 실제로 값이 그 성질을 만족한다고 신뢰할 수 있음
  - 리스트가 비어있지 않다는 성질을 **보장**하는 타입?
- 이러한 성질이 충분히 강력하다면, 테스트나 다른 프로그램 검증 기법 없이도 확신할 수 있음

# 안전한 타입 시스템이란?

안전성은 실행 중에 "잘" 만족해야 하는 성질이 어긋나지 않는 것을 의미

- 그 성질은 언어마다 다름
- 통상적인 타입 시스템은 실행 과정에서 계산된 값이 실제로 타입된 것과 일치하는지를 의미

안전한 타입 시스템이 중요한 이유

- 타입을 통해 원하는 성질을 표현할 수 있다면, 실제로 값이 그 성질을 만족한다고 신뢰할 수 있음
  - 리스트가 비어있지 않다는 성질을 **보장**하는 타입?
- 이러한 성질이 충분히 강력하다면, 테스트나 다른 프로그램 검증 기법 없이도 확신할 수 있음
  - 이것의 극단은 콕 *Coq*, 아그다 *Agda*, 린 *Lean* 등과 같은 값에 기댄 타입 언어 *dependently typed language*

# 조금씩 타입 붙이기 *Gradual typing*

동적인 언어에 타입 시스템을 도입하는 방법

- 안전성에 방점을 두지 않음
- 버그를 잡는데 도움

# 조금씩 타입 붙이기 *Gradual typing*

동적인 언어에 타입 시스템을 도입하는 방법

- 안전성에 방점을 두지 않음
- 버그를 잡는데 도움

**자바스크립트** 타입스크립트, 플로우 *Flow*

**파이썬** 마이파이 *MyPy*, 파이어 *Pyre*, 파이라이트 *Pyright*, 파이타이프 *Pytype*

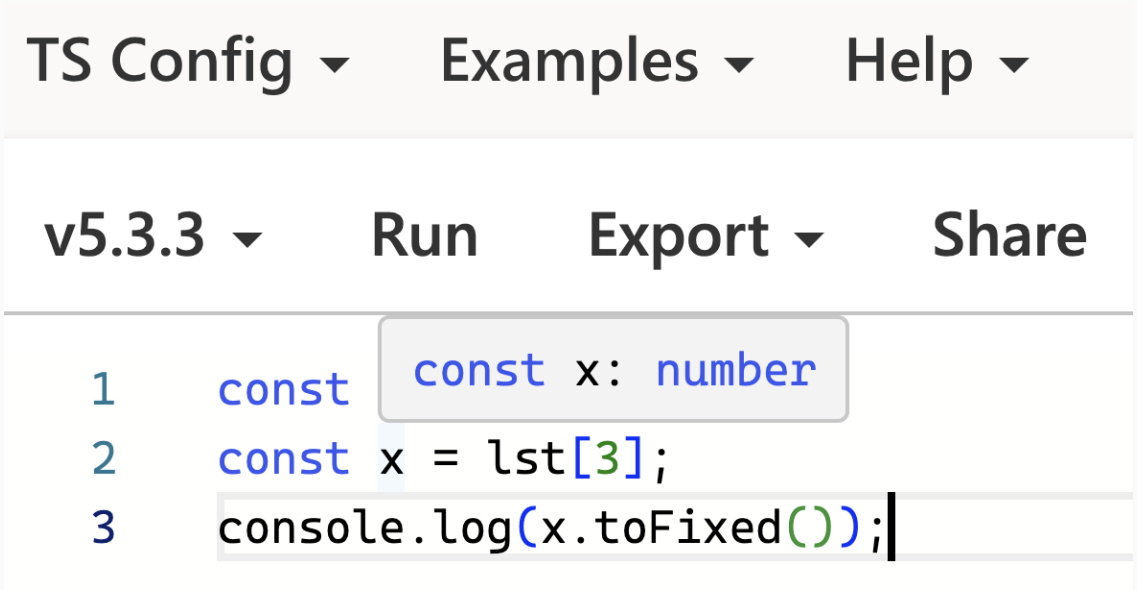
**루비** 소르베 *Sorbet*

**라켓** 타입 라켓 *Typed Racket*

**라쿠** 언어 자체에 내장

# 안전하지 않은 타입 시스템: 타입스크립트

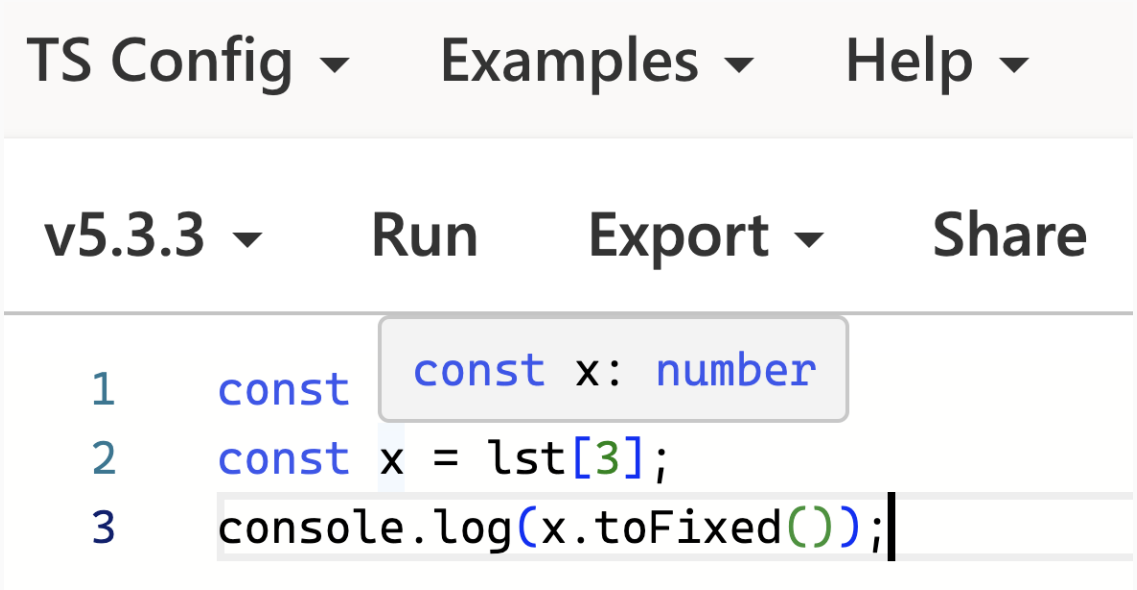
## 타입스크립트는 안전하지 않은 타입 시스템




```
Welcome to Node.js v20.10.0.
Type ".help" for more information.
> const lst = [3, 1, 4];
undefined
> const x = lst[3];  // static type is number but runtime type is undefined.
undefined
> console.log(x.toFixed());
Uncaught TypeError: Cannot read properties of undefined (reading 'toFixed')
```

# 안전하게 조금씩 타입 붙이기

## 타입스크립트는 안전하지 않은 타입 시스템




```
Welcome to Node.js v20.10.0.
Type ".help" for more information.
> const lst = [3, 1, 4];
undefined
> const x = lst[3];  // static type is number but runtime type is undefined.
undefined
> console.log(x.toFixed());
Uncaught TypeError: Cannot read properties of undefined (reading 'toFixed')
```

# 안전하지 않은 타입 시스템: 수동 메모리 관리

수동 메모리 관리를 지원하는 언어는 안전한 타입 시스템을 만들기 굉장히 어려움

```
int toxic(void)
{
        int *x = malloc(sizeof(int));
        *x = 42;
        int *y = x;
        free(x);
        float *z = malloc(sizeof(float));
        *z = 3.141592f;
        return *y + 1;
}
```

# 안전하지 않은 타입 시스템: 수동 메모리 관리

수동 메모리 관리를 지원하는 언어는 안전한 타입 시스템을 만들기 굉장히 어려움

```
int toxic(void)
{
        int *x = malloc(sizeof(int));
        *x = 42;
        int *y = x;
        free(x);
        float *z = malloc(sizeof(float));
        *z = 3.141592f;
        return *y + 1;
}
```

메모리 재활용 *Garbage collection*은 필수적(이었음)

- 러스트는 소유권과 빌려주기의 개념을 사용해 문제를 우회

# 타입 시스템의 장점

## 성능 향상

# 미루지 말고 미리미리

정적 타입 시스템은 실행 전에 타입을 검사하기 때문에,

- 실행 중에 타입 검사를 할 필요가 없고
- 효율적으로 값을 메모리에 담을 수 있음

# 미루지 말고 미리미리

정적 타입 시스템은 실행 전에 타입을 검사하기 때문에,

- 실행 중에 타입 검사를 할 필요가 없고
- 효율적으로 값을 메모리에 담을 수 있음

```
Python 3.11.4 (main, Jul  4 2023, 19:38:47) [Clang 14.0.3 (clang-1403.0.22.14.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 42 + "foo"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

# 타입 시스템의 장점

## 관계 표현

# 간증: 서로 맞물려 돌아가는 *mutually recursive* 구조

https://easyword.kr

# 간증: 서로 맞물려 돌아가는 *mutually recursive* 구조

https://github.com/Zeta611/eko/blob/main/src/Comment.res[1]

```
// Firestore comment entity
@deriving({abstract: light})
type t = {
  @optional id: string,
  content: string,
  user: string,
  timestamp: Firebase.Timestamp.t,
  parent: string,
}
type rec node = {
  comment: t,
  mutable parent: option<node>,
  mutable children: list<node>,
}
let rec countDescendents = children => {
  switch children {
  | list{} => 0
  | list{{children}, ...tl} => 1 + countDescendents(children) + countDescendents(tl)
  }
}
```

[1]영구 링크: https://github.com/Zeta611/eko/...

# 맛보기: 여러 종류의 다형성

**매개변수 다형성** `CollapsibleView`가 받아들이는 `State`, `Control`, `Collapsible` 타입

**타입 적응 다형성** `State` 타입은 `Togglable`의 모양(스위프트에서는 프로토콜 *protocol*이라고 부름)을, `Control`과 `Collapsible` 타입은 `View`의 모양을 가짐

- `CollapsibleView`는 `View`의 모양을 가짐

**아래 타입 다형성** 아래 코드에는 드러나지 않았지만, `Togglable` 타입은 `Equatable`의 아래 타입

`CollapsibleView.swift`

```
struct CollapsibleView<State: Togglable, Control: View, Collapsible: View>: View {
    @Binding var globalState: State
    let state: State
    let control: () -> Control
    let collapsible: () -> Collapsible
    @FocusState private var foo: Bool
    private let val = false

    var body: some View { ... }

    init( ... ) { ... }
}
```

# 타입 시스템과 다형성

# 타입 시스템과 다형성

## 간단한 타입 시스템

# 람다 계산법 *Lambda Calculus*

람다 계산법은 계산의 원리를 연구하기 위해 만들어진 수학적 모델 (Alonzo Church, 1920대)

- 그런데 프로그래밍 언어를 설명하는데 효과적이라는 것이 밝혀짐! (Peter Landin, 1960대)

# 람다 계산법 *Lambda Calculus*

람다 계산법은 계산의 원리를 연구하기 위해 만들어진 수학적 모델 (Alonzo Church, 1920대)

- 그런데 프로그래밍 언어를 설명하는데 효과적이라는 것이 밝혀짐! (Peter Landin, 1960대)

```
Python 3.11.4 (main, Jul  4 2023, 19:38:47) [Clang 14.0.3 (clang-1403.0.22.14.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> fact = lambda n: 1 if n == 0 else n * fact(n - 1)
>>> fact(10)
3628800
```

# 람다 계산법 *Lambda Calculus*

람다 계산법은 계산의 원리를 연구하기 위해 만들어진 수학적 모델 (Alonzo Church, 1920대)

- 그런데 프로그래밍 언어를 설명하는데 효과적이라는 것이 밝혀짐! (Peter Landin, 1960대)

```
Python 3.11.4 (main, Jul  4 2023, 19:38:47) [Clang 14.0.3 (clang-1403.0.22.14.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> fact = lambda n: 1 if n == 0 else n * fact(n - 1)
>>> fact(10)
3628800
```

$$\text{fact} = \lambda\, \text{n}.\ \text{if n = 0 then 1 else n * fact(n - 1)}$$

# 람다 계산법의 문법 구조 *syntax*와 실행 의미 구조 *operational semantics*

## 문법 구조

| | | | | |
|---|---|---|---|---|
| 항 | $t$ | ::= | $x$ | 변수 |
| | | \| | $\lambda x.t$ | 함수 |
| | | \| | $t\ t$ | 적용 |
| 값 | $v$ | ::= | $\lambda x.t$ | 함수 값 |

## 실행 의미 구조

$$t \rightsquigarrow t'$$

E-App1

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}$$

E-App2

$$\frac{t_2 \rightsquigarrow t_2'}{v_1\ t_2 \rightsquigarrow v_1\ t_2'}$$

E-AppAbs

$$\frac{}{(\lambda x.t_{12})\ v_2 \rightsquigarrow [x \mapsto v_2]t_{12}}$$

# 람다 계산법의 문법 구조 *syntax*와 실행 의미 구조 *operational semantics*

## 문법 구조

| | | | | |
|---|---|---|---|---|
| 항 | $t$ | ::= | $x$ | 변수 |
| | | \| | $\lambda x.t$ | 함수 |
| | | \| | $t\ t$ | 적용 |
| 값 | $v$ | ::= | $\lambda x.t$ | 함수 값 |

## 실행 의미 구조

$$t \rightsquigarrow t'$$

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{E-App1} \qquad \frac{t_2 \rightsquigarrow t_2'}{v_1\ t_2 \rightsquigarrow v_1\ t_2'}\ \text{E-App2}$$

$$\frac{}{(\lambda x.t_{12})\ v_2 \rightsquigarrow [x \mapsto v_2]t_{12}}\ \text{E-AppAbs}$$

$$(\lambda x.\lambda y.x)\ \underline{((\lambda x.x)\ a)\ b}$$

$$\xrightsquigarrow{\text{E-App2, E-AppAbs}} \quad \underline{(\lambda x.\lambda y.x)\ a\ b}$$

$$\xrightsquigarrow{\text{E-App1, E-AppAbs}} \quad \underline{(\lambda y.a)\ b}$$

$$\xrightsquigarrow{\text{E-App1}} \quad a$$

# 간단한 타입을 가진 람다 계산법 *Simply Typed Lambda Calculus, STLC*

## 문법 구조

| | | | | |
|---|---|---|---|---|
| 항 | $t$ | ::= | $x$ | 변수 |
| | | \| | $\lambda x{:}\tau.t$ | 함수 |
| | | \| | $t\ t$ | 적용 |
| 값 | $v$ | ::= | $\lambda x{:}\tau.t$ | 함수 값 |
| 타입 | $\tau$ | ::= | $\tau \to \tau$ | 함수 타입 |
| 환경 | $\Gamma$ | ::= | $\emptyset$ | 빈 환경 |
| | | \| | $\Gamma, x : \tau$ | 변수 타입 정의 |

## 타입 규칙

$\boxed{\Gamma \vdash t : \tau}$

T-Var
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

T-Abs
$$\frac{\Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.t_2 : \tau_1 \to \tau_2}$$

T-App
$$\frac{\Gamma \vdash t_1 : \tau_{11} \to \tau_{12} \qquad \Gamma \vdash t_2 : \tau_{11}}{\Gamma \vdash t_1\ t_2 : \tau_{12}}$$

# 타입 규칙 예시

새로운 불리언 항 `true`, `false`를 추가하고, 불리언 타입 `Bool`을 추가:

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{Bool}}\ \textsc{T-True} \qquad \frac{}{\Gamma \vdash \texttt{false} : \texttt{Bool}}\ \textsc{T-False}$$

$$\dfrac{\dfrac{\dfrac{\texttt{x} : \texttt{Bool} \in \texttt{x} : \texttt{Bool}}{\texttt{x} : \texttt{Bool} \vdash \texttt{x} : \texttt{Bool}}\ \textsc{T-Var}}{\vdash \lambda \texttt{x} : \texttt{Bool}.\texttt{x} : \texttt{Bool} \to \texttt{Bool}}\ \textsc{T-Abs} \qquad \dfrac{}{\vdash \texttt{true} : \texttt{Bool}}\ \textsc{T-True}}{\vdash (\lambda \texttt{x} : \texttt{Bool}.\texttt{x})\ \texttt{true} : \texttt{Bool}}\ \textsc{T-App}$$

# 타입 시스템과 다형성

## 매개변수 다형성 *Parametric polymorphism*

# 매개변수 다형성의 동기

인터넷에서 데이터를 가져오는 함수 `fetch`를 만들어보자!

# 매개변수 다형성의 동기

인터넷에서 데이터를 가져오는 함수 `fetch`를 만들어보자!

- 그런데 데이터의 타입마다 다른 함수를 만들어야 하나?

# 매개변수 다형성의 동기

인터넷에서 데이터를 가져오는 함수 `fetch`를 만들어보자!

- 그런데 데이터의 타입마다 다른 함수를 만들어야 하나?

타입에 상관없이 동작하는 일반적인 *generic* 함수를 만들고 싶다!

```
func fetch<T>(
    _ type: T.Type,
    router: Router,
    accessToken: String? = nil
) async throws -> T where T: Decodable {
    logger.debug("fetch(\(type), router: \(router.path), accessToken: \(accessToken ?? "nil")")
    let request = try generateRequest(router: router, accessToken: accessToken)
    let (data, response) = try await session.data(for: request)
... }
```

대충:

$$\texttt{fetch} : \forall T.\texttt{T.Type} \to \texttt{Router} \to \texttt{String?} \to \texttt{T}$$

# 시스템 F

## 문법 구조

| | | | | |
|---|---|---|---|---|
| 항 | $t$ | ::= | $x$ | 변수 |
| | | \| | $\lambda x{:}\tau.t$ | 함수 |
| | | \| | $t\,t$ | 적용 |
| | | \| | $\Lambda X.t$ | 타입 함수 |
| | | \| | $t[\tau]$ | 타입 적용 |
| 값 | $v$ | ::= | $\lambda x{:}\tau.t$ | 함수 값 |
| | | \| | $\Lambda X.t$ | 타입 함수 값 |
| 타입 | $\tau$ | ::= | $X$ | 타입 변수 |
| | | \| | $\tau \to \tau$ | 함수 타입 |
| | | \| | $\forall X.\tau$ | 보편 타입 |
| 환경 | $\Gamma$ | ::= | $\emptyset$ | 빈 환경 |
| | | \| | $\Gamma, x : \tau$ | 변수 타입 정의 |
| | | \| | $\Gamma, X$ | 타입 변수 정의 |

## 실행 의미 구조

$t \rightsquigarrow t'$

$$\frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2}\ \text{E-App1} \qquad \frac{t_2 \rightsquigarrow t_2'}{v_1\ t_2 \rightsquigarrow v_1\ t_2'}\ \text{E-App2}$$

$$\frac{}{(\lambda x.t_{12})\ v_2 \rightsquigarrow [x \mapsto v_2]t_{12}}\ \text{E-AppAbs} \qquad \frac{t_1 \rightsquigarrow t_1'}{t_1[\tau_2] \rightsquigarrow t_1'[\tau_2]}\ \text{E-TApp}$$

$$\frac{}{(\Lambda X.t_{12})[\tau_2] \rightsquigarrow [X \mapsto \tau_2]t_{12}}\ \text{E-TAppTAbs}$$

# 시스템 F

## 문법 구조

| | | | | |
|---|---|---|---|---|
| 항 | $t$ | ::= | $x$ | 변수 |
| | | \| | $\lambda x{:}\tau.t$ | 함수 |
| | | \| | $t\,t$ | 적용 |
| | | \| | $\Lambda X.t$ | 타입 함수 |
| | | \| | $t[\tau]$ | 타입 적용 |
| 값 | $v$ | ::= | $\lambda x{:}\tau.t$ | 함수 값 |
| | | \| | $\Lambda X.t$ | 타입 함수 값 |
| 타입 | $\tau$ | ::= | $X$ | 타입 변수 |
| | | \| | $\tau \to \tau$ | 함수 타입 |
| | | \| | $\forall X.\tau$ | 보편 타입 |
| 환경 | $\Gamma$ | ::= | $\emptyset$ | 빈 환경 |
| | | \| | $\Gamma, x : \tau$ | 변수 타입 정의 |
| | | \| | $\Gamma, X$ | 타입 변수 정의 |

## 타입 규칙

$\boxed{\Gamma \vdash t : \tau}$

T-Var
$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}$$

T-Abs
$$\frac{\Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.t_2 : \tau_1 \to \tau_2}$$

T-App
$$\frac{\Gamma \vdash t_1 : \tau_{11} \to \tau_{12} \quad \Gamma \vdash t_2 : \tau_{11}}{\Gamma \vdash t_1\, t_2 : \tau_{12}}$$

T-TAbs
$$\frac{\Gamma, X \vdash t_2 : \tau_2}{\Gamma \vdash \Lambda X.t_2 : \forall X.\tau_2}$$

T-TApp
$$\frac{\Gamma \vdash t_1 : \forall X.\tau_{12}}{\Gamma \vdash t_1[\tau_2] : [X \mapsto \tau_2]\tau_{12}}$$

# 시스템 F의 예시

```
double = ΛX. λf: X → X. λx: X. f (f x)
> double : ∀X. (X → X) → X → X

doubleInt = double [Int]
> doubleInt : (Int → Int) → Int → Int

doubleIntArrowInt = double [Int → Int]
> doubleIntArrowInt : ((Int → Int) → Int → Int)
                      → (Int → Int) → Int → Int

quadruple = ΛX. double [X → X] (double [X])
> quadruple : ∀X. (X → X) → X → X
```

# let 다형성 *let-polymorphism*

```
OCaml version 5.1.0
Enter #help;; for help.

# let double f a = f (f a);;
val double : ('a -> 'a) -> 'a -> 'a = <fun>
# double ((+) 1) 1;;
- : int = 3
# double ((+.) 1.) 1.;;
- : float = 3.
# let quadruple = double double;;
val quadruple : ('_weak1 -> '_weak1) -> '_weak1 -> '_weak1 = <fun>
# quadruple ((+) 1) 1;;
- : int = 5
# quadruple ((+.) 1.) 1.;;
Error: This expression has type float -> float
       but an expression was expected of type int -> int
       Type float is not compatible with type int
# let quadruple a = double double a;;
val quadruple : ('a -> 'a) -> 'a -> 'a = <fun>
# quadruple ((+) 1) 1;;
- : int = 5
# quadruple ((+.) 1.) 1.;;
- : float = 5.
```

# 타입 시스템과 다형성

## 아래 타입 다형성 *Subtype polymorphism*

# Self의 기묘함

```
Welcome to Apple Swift version 5.9.2 (swiftlang-5.9.2.2.56 clang-1500.1.0.2.5).
Type :help for assistance.
  1> class A {
  2.     let x: Int
  3.     func isEqual(to other: Self) -> Bool {
  4.         self.x == other.x
  5.     }
  6.     init(x: Int) {
  7.         self.x = x
  8.     }
  9. }
error: repl.swift:3:28: error: covariant 'Self' or 'Self?' can only appear as the type of a property,
↪  subscript or method result; did you mean 'A'?
    func isEqual(to other: Self) -> Bool {
                           ^~~~
                           A
```

# Self의 기묘함

```
Welcome to Apple Swift version 5.9.2 (swiftlang-5.9.2.2.56 clang-1500.1.0.2.5).
Type :help for assistance.
  1> class A {
  2.     let x: Int
  3.     func isEqual(to other: Self) -> Bool {
  4.         self.x == other.x
  5.     }
  6.     init(x: Int) {
  7.         self.x = x
  8.     }
  9. }
error: repl.swift:3:28: error: covariant 'Self' or 'Self?' can only appear as the type of a property,
↪  subscript or method result; did you mean 'A'?
    func isEqual(to other: Self) -> Bool {
                           ^~~~
                           A
```

## A를 상속하는 클래스의 `isEqual(to:)`가 이상하다!

# 맞춰 변하기 *covariant*, 거슬러 변하기 *contravariant*, 안 변하기 *invariant*

같은 타입의 다른 값과 같은지 비교하는 함수를 만들자:

```
OCaml version 5.1.0
Enter #help;; for help.
# type x = [ `X ];;
type x = [ `X ]
# type xy = [ `X | `Y ];;
type xy = [ `X | `Y ]
# let x : x = `X;;
val x : x = `X
# let x' = (x :> xy);;
val x' : xy = `X
# let l : x list = [`X; `X];;
val l : x list = [`X; `X]
# let l' = (l :> xy list);;
val l' : xy list = [`X; `X]
# let f : xy -> unit = function `X -> () | `Y -> ();;
val f : xy -> unit = <fun>
# let f' = (f :> x -> unit);;
val f' : x -> unit = <fun>
# let x : x ref = ref `X;;
val x : x ref = {contents = `X}
# let x' = (x :> xy ref);;
Error: Type x ref is not a subtype of xy ref
       The first variant type does not allow tag(s) `Y
```

# 아래 타입 다형성

## 문법 구조

| | | | | |
|---|---|---|---|---|
| 항 | $t$ | ::= | $x$ | 변수 |
| | | $\mid$ | $\lambda x{:}\tau.t$ | 함수 |
| | | $\mid$ | $t\,t$ | 적용 |
| 값 | $v$ | ::= | $\lambda x{:}\tau.t$ | 함수 값 |
| 타입 | $\tau$ | ::= | $\top$ | 최대 타입 |
| | | $\mid$ | $\tau \to \tau$ | 함수 타입 |
| 환경 | $\Gamma$ | ::= | $\emptyset$ | 빈 환경 |
| | | $\mid$ | $\Gamma, x : \tau$ | 변수 타입 정의 |

## 실행 의미 구조

$t \rightsquigarrow t'$

$$\textsc{E-App1}\ \frac{t_1 \rightsquigarrow t_1'}{t_1\ t_2 \rightsquigarrow t_1'\ t_2} \qquad \textsc{E-App2}\ \frac{t_2 \rightsquigarrow t_2'}{v_1\ t_2 \rightsquigarrow v_1\ t_2'} \qquad \textsc{E-AppAbs}\ \frac{}{(\lambda x.t_{12})\ v_2 \rightsquigarrow [x \mapsto v_2]t_{12}}$$

## 아래 타입 규칙

$\varsigma <: \tau$

$$\textsc{S-Refl}\ \frac{}{\varsigma <: \varsigma} \qquad \textsc{S-Trans}\ \frac{\varsigma <: \upsilon \qquad \upsilon <: \tau}{\varsigma <: \tau} \qquad \textsc{S-Top}\ \frac{}{\varsigma <: \top}$$

$$\textsc{S-Arrow}\ \frac{\tau_1 <: \varsigma_1 \qquad \varsigma_2 <: \tau_2}{\varsigma_1 \to \varsigma_2 <: \tau_1 \to \tau_2}$$

## 타입 규칙

$\Gamma \vdash t : \tau$

$$\textsc{T-Var}\ \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \textsc{T-Abs}\ \frac{\Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.t_2 : \tau_1 \to \tau_2}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash t_1 : \tau_{11} \to \tau_{12} \qquad \Gamma \vdash t_2 : \tau_{11}}{\Gamma \vdash t_1\ t_2 : \tau_{12}} \qquad \textsc{T-Sub}\ \frac{\Gamma \vdash t : \varsigma \qquad \varsigma <: \tau}{\Gamma \vdash t : \tau}$$

# 타입 시스템과 다형성

## 타입 적응 다형성 *Ad-hoc polymorphism*

# 타입 적응 다형성

파이썬을 떠올려보면, + 연산자는 정수, 실수, 문자열, 리스트 등 다양한 타입에 대해 다양한 의미를 가짐

# 타입 적응 다형성

파이썬을 떠올려보면, + 연산자는 정수, 실수, 문자열, 리스트 등 다양한 타입에 대해 다양한 의미를 가짐

- C를 떠올려보면, + 연산자를 문자열이나 배열에 대해 사용하는 것은 상상도 못함

# 타입 적응 다형성

파이썬을 떠올려보면, + 연산자는 정수, 실수, 문자열, 리스트 등 다양한 타입에 대해 다양한 의미를 가짐

- C를 떠올려보면, + 연산자를 문자열이나 배열에 대해 사용하는 것은 상상도 못함
- C++에서는 된다!

연산자 및 함수를 같은 이름으로 여러 벌 만드는 오버로딩 *overloading*

```
class X
{
public:
    X& operator+=(const X& rhs) { /* ... */ return *this; }
    friend X operator+(X lhs, const X& rhs)
    {
        lhs += rhs;
        return lhs;
    }
};
```

# 타입 시스템 구멍내기

# 강제로 타입 변환하기

대부분의 언어는 강제로 타입을 변환하는 기능을 제공

- 타입 변환 혹은 메모리 재해석
- FFI *Foreign Function Interface*를 위해서 필요하기도 함

```
> const a = 42 as string
<repl>.ts:4:11 - error TS2352: Conversion of type 'number' to type 'string' may be a mistake because
↪  neither type sufficiently overlaps with the other. If this was intentional, convert the
↪  expression to 'unknown' first.

4 const a = 42 as string
            ~~~~~~~~~~~~

> const a = 42 as unknown as string
undefined
```

# 스위프트의 불투명한 타입 *opaque type*

*Swift provides two ways to hide details about a value's type: opaque types and boxed protocol types. Hiding type information is useful at boundaries between a module and code that calls into the module, because the underlying type of the return value can remain private.*

*A function or method that returns an opaque type hides its return value's type information. Instead of providing a concrete type as the function's return type, the return value is described in terms of the protocols it supports. Opaque types preserve type identity —* ***the compiler has access to the type information, but clients of the module don't.***

—The Swift Programming Language, Opaque Types

# 조심스럽게 타입 변환하기

## ...도 문제다!

```
protocol Color<T> {
  associatedtype T
  var red: T { get }
  var blue: T { get }
  func print(_: T) -> String
}
struct ColorInt: Color {
  let red = 0
  let blue = 1
  func print(_ x: Int) -> String {
    switch x {
    case 0: "red"
    case 1: "blue"
    default: ""
    }
  }
}
```

```
struct ColorBool: Color {
  let red = true
  let blue = false
  func print(_ x: Bool) -> String {
    x ? "red" : "blue"
  }
}
func eqZero<T>(_ x: T) -> Bool {
  guard let x = x as? Int else { return false }
  return x == 0
}
let c1: some Color = ColorInt()
let c2: some Color = ColorBool()
print(c1.print(c1.red))  // "red"
print(c2.print(c2.red))  // "red"
print(eqZero(c1.red))  // true
print(eqZero(c2.red))  // false
```

# 타입 시스템 100% 활용하기

# 타입 시스템 100% 활용하기

## 곱의 합 타입 *Algebraic Data Type, ADT*

# 모든 경우 빠짐 없이 나타내기

```
enum Router {
    case login(LoginParameters)
    case logout(LogoutParameters)
    case refresh(RefreshParameters)
    case me
    case timetableList
    case timetableRead(uuidString: String)
    /* ... */ }
enum Method {
    case get([URLQueryItem])
    case post(Content, [URLQueryItem])
    case patch(Content, [URLQueryItem])
    case delete
    var name: String {
        switch self {
        case .get: return "GET"
        /* ... */ } } }
```

```
var method: Method {
    switch self {
    case let .login(loginParameters):
        guard let data = try?
        ↪  Self.jsonEncoder.encode
        ↪  (loginParameters) else {
            // ...
            return .post(.applicationJSON(nil),
            ↪  [])
        }
        return .post(.applicationJSON(data), [])
/* ... */ } } }
```

switch가 모든 경우를 다 다루고 있음을 컴파일러가 검증

# 저예산 곱의 합 타입 *Poor man's ADT*

```
/* https://github.com/Zeta611/polycalc/blob/main/src/term.h
 * Diagram of the representation for 2xy^2 + 5y + 9 using `TermNode`s
 *  hd   u  next
 * +---+---+---+   +---+---+---+   +---+---+---+
 * | 2 | # | #-+-->| 5 | # | #-+-->| 9 | $ | $ |
 * +---+-|-+---+   +---+-|-+---+   +---+---+---+
 *       |               v
 *       |           +---+---+---+
 *       |           | y | 1 | $ |
 *       |           +---+---+---+
 *       v
 *     +---+---+---+   +---+---+---+
 *     | x | 1 | #-+-->| y | 2 | $ |
 *     +---+---+---+   +---+---+---+
 */
typedef struct TermNode {
        enum { ICOEFF_TERM, RCOEFF_TERM, VAR_TERM } type;
        union {
                long ival;   // ICOEFF_TERM
                double rval; // RCOEFF_TERM
                char *name;  // VAR_TERM
        } hd;
        union {
                struct TermNode *vars; // I/RCOEFF_TERM
                long pow;              // VAR_TERM
        } u;
```

# 재귀적인 곱의 합 타입

다시:

```
// Firestore comment entity
@deriving({abstract: light})
type t = {
  @optional id: string,
  content: string,
  user: string,
  timestamp: Firebase.Timestamp.t,
  parent: string,
}
type rec node = {
  comment: t,
  mutable parent: option<node>,
  mutable children: list<node>,
}
let rec countDescendents = children => {
  switch children {
  | list{} => 0
  | list{{children}, ...tl} => 1 + countDescendents(children) + countDescendents(tl)
  }
}
```

# 타입 시스템 100% 활용하기

## 더 상세한 곱의 합 타입 *Generalized Algebraic Data Type, GADT*

# 곱의 합 타입 다시 생각해보기

```
type 'a t = A of 'a | B of 'a | ...
```

에서, `x : a`라면 각 `A x`, `B x`, …의 타입은 모두 `a t`이다.

# 곱의 합 타입 다시 생각해보기

```
type 'a t = A of 'a | B of 'a | ...
```

에서, `x : a`라면 각 `A x`, `B x`, …의 타입은 모두 `a t`이다.

- A, B, …의 타입은 `'a -> 'a t`

# 곱의 합 타입 다시 생각해보기

```
type 'a t = A of 'a | B of 'a | ...
```

에서, `x : a`라면 각 `A x`, `B x`, …의 타입은 모두 `a t`이다.

- `A`, `B`, …의 타입은 `'a -> 'a t`
- `type 'a t`가 다른 종류의 타입을 가진 구성자 *constructor*들을 모아 놓는다면?

# 비어있지 않은 리스트 표현

곱의 합 타입이 서로 다른 종류의 타입을 가진 구성자들을 모아 놓으면, 타입으로 값들의 구조를 더 잘 좁할 수 있다!

```
type ('a, 'state) list =
| [] : ('a, empty) list
| ( :: ) : 'a * ('a, 'state) list -> ('a, non_empty) list

let hd : type a. (a, non_empty) list -> a = function (x :: _) -> x

let lst1 = []
let lst2 = [3; 1; 4]
let x = hd lst2
(* let y = hd lst1
 *           ^^^^
 * Error: This expression has type ('a, empty) list
 *        but an expression was expected of type ('a, non_empty) list
 *        Type empty is not compatible with type non_empty *)
let y = hd (42 :: lst1)
```

# 타입 시스템 100% 활용하기

## 만능 상자 *Universal type*

# 파이썬처럼 자유롭게?

## 균일하지 않은 리스트를 꼭 써야 될까?

```
Python 3.11.4 (main, Jul  4 2023, 19:38:47) [Clang 14.0.3 (clang-1403.0.22.14.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> l = [0, "hello"]
>>> type(l[0])
<class 'int'>
>>> type(l[1])
<class 'str'>
```

# 파이썬처럼 자유롭게?

## 균일하지 않은 리스트를 꼭 써야 될까?

```
Python 3.11.4 (main, Jul  4 2023, 19:38:47) [Clang 14.0.3 (clang-1403.0.22.14.1)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> l = [0, "hello"]
>>> type(l[0])
<class 'int'>
>>> type(l[1])
<class 'str'>
```

그러나 리액트 *React* 의 `useState`와 같은 함수를 구현하기 위해서는 균일하지 않은 값을 담는 만능 상자가 필요하다.

# 만능 상자로 안전하게!

```
open Base

type univ = exn

module Variant = struct
  let create (type a) () =
    let exception E of a in
    ((fun x -> E x), function E x -> Some x | _
    ↪ -> None)
end
module type S = sig
  type t
  val create : t -> univ
  val match_ : univ -> t option
  val match_exn : univ -> t
end
module Make_univ (T : sig
  type t
end) : S with type t = T.t = struct
  type t = T.t
  let typ = Variant.create ()
  let create x = (fst typ) x
  let match_ x = (snd typ) x
```

```
let match_exn x =
    match match_ x with
    | Some x -> x
    | None -> raise (Invalid_argument
    ↪ "match_exn")
end

module Uint = Make_univ (Int)
module Ustring = Make_univ (String)

let u1 = Uint.create 0
let u2 = Ustring.create "hello"
let l = [u1; u2]

let _ = Uint.match_ (List.nth_exn l 0)
let _ = Uint.match_ (List.nth_exn l 1)
let _ = Ustring.match_ (List.nth_exn l 0)
let _ = Ustring.match_ (List.nth_exn l 1)
```

# 타입 시스템 100% 활용하기

## 러스트 *Rust*의 아핀 타입 *Affine type*으로 자원 안전하게

# 공유된 상태는 에러의 원인

변할 수 있는 상태를 공유하는 것은 굉장히 위험한 일!

```c
int *x = malloc(sizeof(int));
*x = 42;
int *y = x;
free(x);
free(y); // double-free
printf("%d\n", *x); // use-after-free
```

# 메모리 버그를 타입 시스템으로 잡자

## 참조 *Reference*의 규칙[2]

- 항상 변할 수 있는 참조 하나 혹은 임의의 개수의 참조들만 존재해야 한다 (둘 중 하나).
- 참조는 항상 유효해야 한다.

```
fn foo(x: &mut i32, y: &mut i32) -> i32 {
  *x = 42; *y = 0;
  return *x; // 42 보장
}
```

타입 시스템이 x가 가리키는 값의 **주인**과 y가 가리키는 값의 **주인**이 다르다고 보장

[2] The Rust Programming Language, References and Borrowing

# 모나드 *Monad*와 효과 시스템 *effect system*으로 부수 효과 안전하게

모나드 발명하기 세미나 자료 참고