

Typing Dynamic Languages

Polymorphism, subtyping, typecases and gradual typing

Mickaël Laurent

2026

FIT, Czech Technical University

Dynamic languages

Python, JavaScript, R, Lua, Elixir, ...

Dynamic languages

Python, JavaScript, R, Lua, Elixir, ...

- Manipulation of **heterogeneous data** (heterogeneous lists, JSON, ...)

```
l = [1, "two", 3.0, None, [4, 5]]
```

```
user = json.loads(s)
```

```
# {"name": "Alice", "age": 42, "tags": ["a", 1]}
```

```
print(user["name"] + " is " + str(user["age"]) + " years old")
```

Dynamic languages

Python, JavaScript, R, Lua, Elixir, ...

- Manipulation of **heterogeneous data** (heterogeneous lists, JSON, ...)
- **Dynamic loading** of data and code

```
config = json.load(open("config.json"))  
plugin = importlib.import_module(config["plugin"])  
plugin.run(*config["args"])  
  
e = eval(input("Expression: "))
```

Dynamic languages

Python, JavaScript, R, Lua, Elixir, ...

- Manipulation of **heterogeneous data** (heterogeneous lists, JSON, ...)
- **Dynamic loading** of data and code
- Runtime **type tests** (`isinstance`, `typeof`, ...)

```
def to_str(x):  
    if isinstance(x, int):  
        return str(x)  
    else:  
        return x
```

Dynamic languages

Python, JavaScript, R, Lua, Elixir, ...

- Manipulation of **heterogeneous data** (heterogeneous lists, JSON, ...)
- **Dynamic loading** of data and code
- Runtime **type tests** (`isinstance`, `typeof`, ...)
- **Overloading**: the same code, different behaviours depending on argument types

```
def to_str(x):  
    if isinstance(x, int):  
        return str(x)  
    else:  
        return x
```

Dynamic languages

Python, JavaScript, R, Lua, Elixir, ...

- Manipulation of **heterogeneous data** (heterogeneous lists, JSON, ...)
- **Dynamic loading** of data and code
- Runtime **type tests** (`isinstance`, `typeof`, ...)
- **Overloading**: the same code, different behaviours depending on argument types

```
def to_str(x):  
    if isinstance(x, int):  
        return str(x)  
    else:  
        return x
```

What type should `to_str` have?

Convenient idioms: population of a city (CSV)

Rust:

```
fn get_population(data: &str, city: &str) -> Option<u32> {  
    let mut rdr = csv::Reader::from_reader(data.as_bytes());  
    let h = rdr.headers().unwrap().clone();  
    let ci = h.iter().position(|x| x == "city").unwrap();  
    let pi = h.iter().position(|x| x == "pop").unwrap();  
    for r in rdr.records() {  
        let r = r.unwrap();  
        if r.get(ci).unwrap() == city {  
            return Some(r.get(pi).unwrap().parse().unwrap());  
        }  
    }  
    None  
}
```

Python:

```
def get_population(data, city):  
    rows = csv.DictReader(StringIO(data))  
    for row in rows:  
        if row["city"] == city:  
            return int(row["pop"])  
    return None
```

Convenient idioms: population of a city (CSV)

Rust:

```
fn get_population(data: &str, city: &str) -> Option<u32> {  
    let mut rdr = csv::Reader::from_reader(data.as_bytes());  
    let h = rdr.headers().unwrap().clone();  
    let ci = h.iter().position(|x| x == "city").unwrap();  
    let pi = h.iter().position(|x| x == "pop").unwrap();  
    for r in rdr.records() {  
        let r = r.unwrap();  
        if r.get(ci).unwrap() == city {  
            return Some(r.get(pi).unwrap().parse().unwrap());  
        }  
    }  
    None  
}
```

Python:

```
def get_population(data, city):  
    rows = csv.DictReader(StringIO(data))  
    for row in rows:  
        if row["city"] == city:  
            return int(row["pop"])  
    return None
```

Convenient idioms: population of a city (CSV)

Rust:

```
fn get_population(data: &str, city: &str) -> Option<u32> {  
    let mut rdr = csv::Reader::from_reader(data.as_bytes());  
    let h = rdr.headers().unwrap().clone();  
    let ci = h.iter().position(|x| x == "city").unwrap();  
    let pi = h.iter().position(|x| x == "pop").unwrap();  
    for r in rdr.records() {  
        let r = r.unwrap();  
        if r.get(ci).unwrap() == city {  
            return Some(r.get(pi).unwrap().parse().unwrap());  
        }  
    }  
    None  
}
```

```
get_population(data, "Gif-sur-Yvette").unwrap() / 1000 // Rust: 22  
get_population(data, "Gif-sur-Yvette") // 1000        # Python: 22
```

Python:

```
def get_population(data, city):  
    rows = csv.DictReader(StringIO(data))  
    for row in rows:  
        if row["city"] == city:  
            return int(row["pop"])  
    return None
```

Typing a dynamic language

- Increased flexibility and conciseness
 - ✓ More readable code, easier experimentation
 - ✗ Harder to scale
- (less explicit code, little/no type information, etc.)

Typing a dynamic language

- Increased flexibility and conciseness
 - ✓ More readable code, easier experimentation
 - ✗ Harder to scale
(less explicit code, little/no type information, etc.)
- These idioms rely on **untagged unions** (`int` or `None`), **type tests**, **heterogeneous data**, ...

Typing a dynamic language

- Increased flexibility and conciseness
 - ✓ More readable code, easier experimentation
 - ✗ Harder to scale
(less explicit code, little/no type information, etc.)
- These idioms rely on **untagged unions** (`int` or `None`), **type tests**, **heterogeneous data**, ...
- The type checker must accept the **idioms** of the language
⇒ we cannot restrict the language (unlike OCaml, Rust, ...)

Static type checkers for dynamic languages

Language	Type checker	Subtyping	Remarks
JavaScript	TypeScript	algebraic (not formalized)	most widely used, unsound Meta
	Flow	algebraic (not formalized)	
Python	MyPy, Pyright	algebraic (not formalized)	PEP 484 intersections, negations
	Ty	algebraic (not formalized)	
Lua	Luau	semantic (almost) [Jeffrey, 2022]	Roblox
Elixir	built-in	semantic [Castagna et al., 2023]	not polymorphic yet
Erlang	Etylizer	semantic [Schimpf et al., 2023]	research prototype
MLsem	built-in	semantic [Laurent and Vitek, 2026]	research prototype

Why type a dynamic language?

- Detect **errors** before execution (**None**, missing field, ...)
- **Document** the code, with machine-checked documentation
- **Tooling**: autocompletion, navigation, refactoring
- Scaling up: from scripts to **large codebases** (Dropbox, Instagram, ...)
- Improve performance by enabling new optimizations?

Why type a dynamic language?

- Detect **errors** before execution (**None**, missing field, ...)
- **Document** the code, with machine-checked documentation
- **Tooling**: autocompletion, navigation, refactoring
- Scaling up: from scripts to **large codebases** (Dropbox, Instagram, ...)
- Improve performance by enabling new optimizations?

In this lecture:

- Different kinds of polymorphism (parametric, unions, intersections)
- Subtyping (semantic vs algebraic)
- Typecases, type narrowing and overloading
- Gradual typing

Different kinds of polymorphism

Different kinds of polymorphism

Type algebra and subtyping

Typecases, type narrowing and overloading

Gradual typing

Parametric polymorphism

```
def first(p):  
    x, y = p  
    return x
```

Which type?

Parametric polymorphism

```
def first(p):  
    x, y = p  
    return x
```

Which type?

$$\forall \alpha, \beta. (\alpha \times \beta) \rightarrow \alpha$$

Parametric polymorphism

```
def first(p):
```

```
    x, y = p
```

```
    return x
```

Which type?

$$\forall \alpha, \beta. (\alpha \times \beta) \rightarrow \alpha$$

- **Parametric polymorphism** (= *generics*): the same code works **uniformly** for all types (seen in class: Hindley-Milner)

Parametric polymorphism

```
def first(p):
```

```
    x, y = p
```

```
    return x
```

Which type?

$$\forall \alpha, \beta. (\alpha \times \beta) \rightarrow \alpha$$

- **Parametric polymorphism** (= *generics*): the same code works **uniformly** for all types (seen in class: Hindley-Milner)
- α, β : **type variables**, instantiated at each use
e.g. `first((1, "a")): int` (with $\alpha := \text{int}$, $\beta := \text{string}$)

Subtype polymorphism

```
def get_population(data, city):  
    rows = csv.DictReader(StringIO(data))  
    for row in rows:  
        if row["city"] == city:  
            return int(row["pop"])  
    return None
```

Which type?

Subtype polymorphism

```
def get_population(data, city):  
    rows = csv.DictReader(StringIO(data))  
    for row in rows:  
        if row["city"] == city:  
            return int(row["pop"])  
    return None
```

Which type?

$(\text{string} \times \text{string}) \rightarrow \text{int} \vee \text{none}$

- **Union** type: expresses several possibilities

Subtype polymorphism

```
def get_population(data, city):  
    rows = csv.DictReader(StringIO(data))  
    for row in rows:  
        if row["city"] == city:  
            return int(row["pop"])  
    return None
```

Which type?

$(\text{string} \times \text{string}) \rightarrow \text{int} \vee \text{none}$

- **Union** type: expresses **several possibilities**
- `int` is a **subtype** of `int ∨ none`
⇒ if a value has type `int`, then it also has type `int ∨ none`

Subtype polymorphism

```
def get_population(data, city):  
    rows = csv.DictReader(StringIO(data))  
    for row in rows:  
        if row["city"] == city:  
            return int(row["pop"])  
    return None
```

Which type?

$(\text{string} \times \text{string}) \rightarrow \text{int} \vee \text{none}$

- **Union** type: expresses **several possibilities**
- `int` is a **subtype** of `int ∨ none`
⇒ if a value has type `int`, then it also has type `int ∨ none`
- **Subtype polymorphism**: the same value can have several types

Sum types vs *true* union types

	Sum types (ADT)	Union types
Languages	OCaml, Haskell, Rust <code>type t = I of int S of string</code>	TypeScript, Python, Luau <code>int str</code>

Sum types vs *true* union types

	Sum types (ADT)	Union types
Languages	OCaml, Haskell, Rust <code>type t = I of int S of string</code>	TypeScript, Python, Lua <code>int str</code>
Values	Tags $\Rightarrow$ new values (<code>I 3</code>), disjoint from those of other types	No tag $\Rightarrow$ arbitrary values (<code>3</code> , <code>"foo"</code>)

Sum types vs *true* union types

	Sum types (ADT)	Union types
Languages	OCaml, Haskell, Rust <code>type t = I of int S of string</code>	TypeScript, Python, Lua <code>int str</code>
Values	Tags $\Rightarrow$ new values (<code>I 3</code>), disjoint from those of other types	No tag $\Rightarrow$ arbitrary values (<code>3</code> , <code>"foo"</code>)
Subtyping	None (disjoint types) <code>3 : t</code> rejected , one must write <code>I 3</code>	Required <code>int $\leq$ int $\vee$ string $\leq$ int $\vee$ string $\vee$ none</code>

Ad-hoc polymorphism

Which type?

```
def neg(x):  
    if isinstance(x, bool):  
        return not x  
    else:  
        return -x
```

Ad-hoc polymorphism

```
def neg(x):  
    if isinstance(x, bool):  
        return not x  
    else:  
        return -x
```

Which type?

$(\text{int} \vee \text{bool}) \rightarrow (\text{int} \vee \text{bool})$

Correct, but **imprecise**:

`neg(3) + 1` would be rejected!

Ad-hoc polymorphism

```
def neg(x):  
    if isinstance(x, bool):  
        return not x  
    else:  
        return -x
```

Which type?

$(\text{int} \vee \text{bool}) \rightarrow (\text{int} \vee \text{bool})$

Correct, but **imprecise**:

`neg(3) + 1` would be rejected!

$(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$

Ad-hoc polymorphism

```
def neg(x):  
    if isinstance(x, bool):  
        return not x  
    else:  
        return -x
```

Which type?

$(\text{int} \vee \text{bool}) \rightarrow (\text{int} \vee \text{bool})$

Correct, but **imprecise**:

`neg(3) + 1` would be rejected!

$(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$

- **Intersection** of arrows: the function implements **both behaviours**

Ad-hoc polymorphism

```
def neg(x):  
    if isinstance(x, bool):  
        return not x  
    else:  
        return -x
```

Which type?

$(\text{int} \vee \text{bool}) \rightarrow (\text{int} \vee \text{bool})$

Correct, but **imprecise**:

`neg(3) + 1` would be rejected!

$(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$

- **Intersection** of arrows: the function implements **both behaviours**
- **Ad-hoc polymorphism**: the same code, a **different** behaviour depending on the type of the argument (**overloading**)

Ad-hoc polymorphism

```
def neg(x):
```

```
    if isinstance(x, bool):
```

```
        return not x
```

```
    else:
```

```
        return -x
```

Which type?

$(\text{int} \vee \text{bool}) \rightarrow (\text{int} \vee \text{bool})$

Correct, but **imprecise**:

`neg(3) + 1` would be rejected!

$(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$

- **Intersection** of arrows: the function implements **both behaviours**
- **Ad-hoc polymorphism**: the same code, a **different** behaviour depending on the type of the argument (**overloading**)
- **Typeclasses** (Haskell, Rust traits) also provide ad-hoc polymorphism, but:
 - the choice is made on the type of **a single** parameter (the object, `self`)
 - each type must **explicitly declare** its methods (`impl Neg for i32 {...}`)

Three forms of polymorphism

Polymorphism	Example	Idea
Parametric	$\forall \alpha, \beta. (\alpha \times \beta) \rightarrow \alpha$	same code, for all types

Three forms of polymorphism

Polymorphism	Example	Idea
Parametric	$\forall \alpha, \beta. (\alpha \times \beta) \rightarrow \alpha$	same code, for all types
Subtyping (unions)	<code>int ≤ int ∨ none</code>	a value has several types

Three forms of polymorphism

Polymorphism	Example	Idea
Parametric	$\forall \alpha, \beta. (\alpha \times \beta) \rightarrow \alpha$	same code, for all types
Subtyping (unions)	<code>int ≤ int ∨ none</code>	a value has several types
Ad-hoc (intersections)	<code>(int → int) ∧ (bool → bool)</code>	behaviour depends on the type

Three forms of polymorphism

Polymorphism	Example	Idea
Parametric	$\forall \alpha, \beta. (\alpha \times \beta) \rightarrow \alpha$	same code, for all types
Subtyping (unions)	<code>int ≤ int ∨ none</code>	a value has several types
Ad-hoc (intersections)	<code>(int → int) ∧ (bool → bool)</code>	behaviour depends on the type

Dynamic languages use **all three** (often at the same time)!

⇒ We need a type algebra that combines them.

Type algebra and subtyping

Different kinds of polymorphism

Type algebra and subtyping

Typecases, type narrowing and overloading

Gradual typing

Set-theoretic types

Set-theoretic types $t ::= b \mid \alpha \mid t \rightarrow t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \text{any} \mid \text{empty}$

- b : base types (int, bool, ...)

Set-theoretic types

Set-theoretic types $t ::= b \mid \alpha \mid t \rightarrow t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \text{any} \mid \text{empty}$

- b : base types (int, bool, ...)
- **Type variables** α : parametric polymorphism (e.g. $\forall \alpha. \alpha \rightarrow \alpha$)

Set-theoretic types

Set-theoretic types $t ::= b \mid \alpha \mid t \rightarrow t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \text{any} \mid \text{empty}$

- b : base types (`int`, `bool`, ...)
- **Type variables** α : parametric polymorphism (e.g. $\forall \alpha. \alpha \rightarrow \alpha$)
- **Union** $t_1 \vee t_2$: to express possibilities (e.g. `int  $\vee$  none`)

Set-theoretic types

Set-theoretic types $t ::= b \mid \alpha \mid t \rightarrow t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \text{any} \mid \text{empty}$

- b : base types (`int`, `bool`, ...)
- **Type variables** α : parametric polymorphism (e.g. $\forall \alpha. \alpha \rightarrow \alpha$)
- **Union** $t_1 \vee t_2$: to express possibilities (e.g. `int` $\vee$ `none`)
- **Intersection** $t_1 \wedge t_2$: overloading (e.g. $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$),
or to combine *traits* (e.g. `printable` $\wedge$ `iterable`)

Set-theoretic types

Set-theoretic types $t ::= b \mid \alpha \mid t \rightarrow t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \text{any} \mid \text{empty}$

- b : base types (`int`, `bool`, ...)
- **Type variables** α : parametric polymorphism (e.g. $\forall \alpha. \alpha \rightarrow \alpha$)
- **Union** $t_1 \vee t_2$: to express possibilities (e.g. `int` $\vee$ `none`)
- **Intersection** $t_1 \wedge t_2$: overloading (e.g. $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$), or to combine *traits* (e.g. `printable` $\wedge$ `iterable`)
- **Negation** $\neg t$: allows expressing difference
(e.g. $t_1 \setminus t_2 \equiv t_1 \wedge \neg t_2$: everything of type t_1 but not of type t_2)

Set-theoretic types

Set-theoretic types $t ::= b \mid \alpha \mid t \rightarrow t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \text{any} \mid \text{empty}$

- b : base types (`int`, `bool`, ...)
- **Type variables** α : parametric polymorphism (e.g. $\forall \alpha. \alpha \rightarrow \alpha$)
- **Union** $t_1 \vee t_2$: to express possibilities (e.g. `int` $\vee$ `none`)
- **Intersection** $t_1 \wedge t_2$: overloading (e.g. $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$), or to combine *traits* (e.g. `printable` $\wedge$ `iterable`)
- **Negation** $\neg t$: allows expressing difference
(e.g. $t_1 \setminus t_2 \equiv t_1 \wedge \neg t_2$: everything of type t_1 but not of type t_2)
- **any**: all values have this type

Set-theoretic types

Set-theoretic types $t ::= b \mid \alpha \mid t \rightarrow t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \text{any} \mid \text{empty}$

- b : base types (`int`, `bool`, ...)
- **Type variables** α : parametric polymorphism (e.g. $\forall \alpha. \alpha \rightarrow \alpha$)
- **Union** $t_1 \vee t_2$: to express possibilities (e.g. `int` $\vee$ `none`)
- **Intersection** $t_1 \wedge t_2$: overloading (e.g. $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$), or to combine *traits* (e.g. `printable` $\wedge$ `iterable`)
- **Negation** $\neg t$: allows expressing difference
(e.g. $t_1 \setminus t_2 \equiv t_1 \wedge \neg t_2$: everything of type t_1 but not of type t_2)
- **any**: all values have this type
- **empty**: no value has this type (but a diverging expression does)

Subtyping

- A function that accepts an argument of type `int ∨ bool` should also accept an argument of type `int`

Subtyping

- A function that accepts an argument of type $\text{int} \vee \text{bool}$ should also accept an argument of type int
- Formally: $\text{int} \leq \text{int} \vee \text{bool}$
(int is a **subtype** of $\text{int} \vee \text{bool}$, and $\text{int} \vee \text{bool}$ a **supertype** of int)

Subtyping

- A function that accepts an argument of type $\text{int} \vee \text{bool}$ should also accept an argument of type int
- Formally: $\text{int} \leq \text{int} \vee \text{bool}$
(int is a **subtype** of $\text{int} \vee \text{bool}$, and $\text{int} \vee \text{bool}$ a **supertype** of int)
- $t_1 \leq t_2$: an expression of type t_1 can be used wherever an expression of type t_2 is expected

Subtyping

- A function that accepts an argument of type $\text{int} \vee \text{bool}$ should also accept an argument of type int
- Formally: $\text{int} \leq \text{int} \vee \text{bool}$
(int is a **subtype** of $\text{int} \vee \text{bool}$, and $\text{int} \vee \text{bool}$ a **supertype** of int)
- $t_1 \leq t_2$: an expression of type t_1 can be used wherever an expression of type t_2 is expected
- $\leq$ is an **order relation**:
 - reflexive ($t \leq t$)
 - transitive ($t_1 \leq t_2$ and $t_2 \leq t_3 \Rightarrow t_1 \leq t_3$)
 - antisymmetric ($t_1 \leq t_2$ and $t_2 \leq t_1 \Rightarrow t_1 \simeq t_2$)

Algebraic subtyping

Algebraic approach: $\leq$ is defined by inference rules.

$$[\text{Refl}] \frac{}{t \leq t}$$

$$[\text{Trans}] \frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3}$$

$$[\text{Arrow}] \frac{s_2 \leq s_1 \quad t_1 \leq t_2}{s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2}$$

$$[\text{Pair}] \frac{s_1 \leq s_2 \quad t_1 \leq t_2}{s_1 \times t_1 \leq s_2 \times t_2}$$

Algebraic subtyping

Algebraic approach: $\leq$ is defined by inference rules.

$$[\text{Refl}] \frac{}{t \leq t}$$

$$[\text{Trans}] \frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3}$$

$$[\text{Arrow}] \frac{s_2 \leq s_1 \quad t_1 \leq t_2}{s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2}$$

$$[\text{Pair}] \frac{s_1 \leq s_2 \quad t_1 \leq t_2}{s_1 \times t_1 \leq s_2 \times t_2}$$

$$[\text{Empty}] \frac{}{\text{empty} \leq t}$$

$$[\text{Any}] \frac{}{t \leq \text{any}}$$

$$[\vee L] \frac{t_1 \leq t \quad t_2 \leq t}{t_1 \vee t_2 \leq t}$$

$$[\vee R] \frac{t \leq t_i}{t \leq t_1 \vee t_2}$$

$$[\wedge L] \frac{t_i \leq t}{t_1 \wedge t_2 \leq t}$$

$$[\wedge R] \frac{t \leq t_1 \quad t \leq t_2}{t \leq t_1 \wedge t_2}$$

Algebraic subtyping

Algebraic approach: $\leq$ is defined by inference rules.

$$[\text{Refl}] \frac{}{t \leq t}$$

$$[\text{Trans}] \frac{t_1 \leq t_2 \quad t_2 \leq t_3}{t_1 \leq t_3}$$

$$[\text{Arrow}] \frac{s_2 \leq s_1 \quad t_1 \leq t_2}{s_1 \rightarrow t_1 \leq s_2 \rightarrow t_2}$$

$$[\text{Pair}] \frac{s_1 \leq s_2 \quad t_1 \leq t_2}{s_1 \times t_1 \leq s_2 \times t_2}$$

$$[\text{Empty}] \frac{}{\text{empty} \leq t}$$

$$[\text{Any}] \frac{}{t \leq \text{any}}$$

$$[\vee L] \frac{t_1 \leq t \quad t_2 \leq t}{t_1 \vee t_2 \leq t}$$

$$[\vee R] \frac{t \leq t_i}{t \leq t_1 \vee t_2}$$

$$[\wedge L] \frac{t_i \leq t}{t_1 \wedge t_2 \leq t}$$

$$[\wedge R] \frac{t \leq t_1 \quad t \leq t_2}{t \leq t_1 \wedge t_2}$$

Simple, fast, easy to implement (e.g. MLsub, MLscript).

But **negation** $\neg t$ is hard to handle with this kind of rules.

Algebraic subtyping: an example

$$(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \stackrel{?}{\leq} \text{int} \times (\text{bool} \vee \text{int})$$

Algebraic subtyping: an example

$$(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \stackrel{?}{\leq} \text{int} \times (\text{bool} \vee \text{int})$$

Yes: we build a derivation (bottom-up, the leaves are [Refl]).

$$(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \leq \text{int} \times (\text{bool} \vee \text{int})$$

Algebraic subtyping: an example

$$(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \stackrel{?}{\leq} \text{int} \times (\text{bool} \vee \text{int})$$

Yes: we build a derivation (bottom-up, the leaves are [Ref]).

$$[\vee L] \frac{\text{int} \times \text{bool} \leq \text{int} \times (\text{bool} \vee \text{int}) \quad \text{int} \times \text{int} \leq \text{int} \times (\text{bool} \vee \text{int})}{(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \leq \text{int} \times (\text{bool} \vee \text{int})}$$

Algebraic subtyping: an example

$$(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \stackrel{?}{\leq} \text{int} \times (\text{bool} \vee \text{int})$$

Yes: we build a derivation (bottom-up, the leaves are [Refl]).

$$\begin{array}{c} \text{[Pair]} \frac{\text{int} \leq \text{int} \quad \text{bool} \leq \text{bool} \vee \text{int}}{\text{int} \times \text{bool} \leq \text{int} \times (\text{bool} \vee \text{int})} \quad \text{[Pair]} \frac{\text{int} \leq \text{int} \quad \text{int} \leq \text{bool} \vee \text{int}}{\text{int} \times \text{int} \leq \text{int} \times (\text{bool} \vee \text{int})} \\ \text{[\vee L]} \frac{}{(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \leq \text{int} \times (\text{bool} \vee \text{int})} \end{array}$$

Algebraic subtyping: an example

$$(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \stackrel{?}{\leq} \text{int} \times (\text{bool} \vee \text{int})$$

Yes: we build a derivation (bottom-up, the leaves are [Refl]).

$$\begin{array}{c} \text{[Pair]} \frac{\text{int} \leq \text{int} \quad \text{[}\vee\text{R]} \frac{\text{bool} \leq \text{bool}}{\text{bool} \leq \text{bool} \vee \text{int}}}{\text{int} \times \text{bool} \leq \text{int} \times (\text{bool} \vee \text{int})} \quad \text{[Pair]} \frac{\text{int} \leq \text{int} \quad \text{[}\vee\text{R]} \frac{\text{int} \leq \text{int}}{\text{int} \leq \text{bool} \vee \text{int}}}{\text{int} \times \text{int} \leq \text{int} \times (\text{bool} \vee \text{int})} \\ \text{[}\vee\text{L]} \frac{}{(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \leq \text{int} \times (\text{bool} \vee \text{int})} \end{array}$$

Algebraic subtyping: an example

$$(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \stackrel{?}{\leq} \text{int} \times (\text{bool} \vee \text{int})$$

Yes: we build a derivation (bottom-up, the leaves are [Refl]).

$$\begin{array}{c} \text{[Pair]} \frac{\frac{\text{int} \leq \text{int}}{\text{int} \times \text{bool} \leq \text{int} \times (\text{bool} \vee \text{int})} \quad \text{[}\vee\text{R]} \frac{\frac{\text{bool} \leq \text{bool}}{\text{bool} \leq \text{bool} \vee \text{int}}}{\text{int} \times \text{bool} \leq \text{int} \times (\text{bool} \vee \text{int})} \quad \text{[Pair]} \frac{\frac{\text{int} \leq \text{int}}{\text{int} \times \text{int} \leq \text{int} \times (\text{bool} \vee \text{int})} \quad \text{[}\vee\text{R]} \frac{\frac{\text{int} \leq \text{int}}{\text{int} \leq \text{bool} \vee \text{int}}}{\text{int} \times \text{int} \leq \text{int} \times (\text{bool} \vee \text{int})} \\ \text{[}\vee\text{L]} \frac{}{(\text{int} \times \text{bool}) \vee (\text{int} \times \text{int}) \leq \text{int} \times (\text{bool} \vee \text{int})} \end{array}$$

Exercise

Using the rules of algebraic subtyping, derive the following relations, or explain why it is impossible:

1. $\text{int} \wedge \text{bool} \leq \text{bool} \vee \text{string}$
2. $\text{int} \times \text{bool} \leq (\text{int} \vee \text{bool}) \times \text{any}$
3. $(\text{int} \vee \text{bool}) \rightarrow \text{int} \leq \text{int} \rightarrow (\text{int} \vee \text{bool})$
4. $\text{int} \rightarrow \text{int} \leq (\text{int} \vee \text{bool}) \rightarrow \text{int}$
5. $\text{any} \rightarrow \text{empty} \leq \text{empty} \rightarrow \text{any}$ (and the other way around?)

Limits of algebraic subtyping

$$\text{int} \times (\text{bool} \vee \text{int}) \stackrel{?}{\leq} (\text{int} \times \text{bool}) \vee (\text{int} \times \text{int})$$

Limits of algebraic subtyping

$$\text{int} \times (\text{bool} \vee \text{int}) \stackrel{?}{\leq} (\text{int} \times \text{bool}) \vee (\text{int} \times \text{int})$$

Subtyping not derivable: $[\vee R]$ forces us to pick *one* member of the union...

Limits of algebraic subtyping

$$\text{int} \times (\text{bool} \vee \text{int}) \stackrel{?}{\leq} (\text{int} \times \text{bool}) \vee (\text{int} \times \text{int})$$

Subtyping not derivable: $[\vee R]$ forces us to pick *one* member of the union...

$$(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool}) \stackrel{?}{\leq} (\text{int} \vee \text{bool}) \rightarrow (\text{int} \vee \text{bool})$$

Limits of algebraic subtyping

$$\text{int} \times (\text{bool} \vee \text{int}) \stackrel{?}{\leq} (\text{int} \times \text{bool}) \vee (\text{int} \times \text{int})$$

Subtyping not derivable: $[\vee R]$ forces us to pick *one* member of the union...

$$(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool}) \stackrel{?}{\leq} (\text{int} \vee \text{bool}) \rightarrow (\text{int} \vee \text{bool})$$

Subtyping not derivable: $[\wedge L]$ forces us to pick *one* member of the intersection...

$\Rightarrow$ Intersections of arrows are useless $\Rightarrow$ No ad-hoc polymorphism

Limits of algebraic subtyping

$$\text{int} \times (\text{bool} \vee \text{int}) \stackrel{?}{\leq} (\text{int} \times \text{bool}) \vee (\text{int} \times \text{int})$$

Subtyping not derivable: $[\vee R]$ forces us to pick *one* member of the union...

$$(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool}) \stackrel{?}{\leq} (\text{int} \vee \text{bool}) \rightarrow (\text{int} \vee \text{bool})$$

Subtyping not derivable: $[\wedge L]$ forces us to pick *one* member of the intersection...

$\Rightarrow$ Intersections of arrows are useless $\Rightarrow$ No ad-hoc polymorphism

We can add more rules (distributivity, ...), but where do we stop?

$\Rightarrow$ We need a reference **definition** for $\leq$.

Semantic subtyping [Frisch et al., 2008]

Idea: interpret a type as the **set of values** it contains: $\llbracket t \rrbracket \subseteq \mathcal{V}$

$$t_1 \leq t_2 \stackrel{\text{def}}{\iff} \llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket$$

Semantic subtyping [Frisch et al., 2008]

Idea: interpret a type as the **set of values** it contains: $\llbracket t \rrbracket \subseteq \mathcal{V}$

$$t_1 \leq t_2 \stackrel{\text{def}}{\iff} \llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket$$

$\llbracket \text{int} \rrbracket$	$=$	$\mathbb{Z}$	$\llbracket t_1 \vee t_2 \rrbracket$	$=$	$\llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$
$\llbracket \text{bool} \rrbracket$	$=$	$\{\text{true}, \text{false}\}$	$\llbracket t_1 \wedge t_2 \rrbracket$	$=$	$\llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$
$\llbracket \text{any} \rrbracket$	$=$	$\mathcal{V}$	$\llbracket \neg t \rrbracket$	$=$	$\mathcal{V} \setminus \llbracket t \rrbracket$
$\llbracket \text{empty} \rrbracket$	$=$	$\emptyset$	$\llbracket t_1 \times t_2 \rrbracket$	$=$	$\llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$

Semantic subtyping [Frisch et al., 2008]

Idea: interpret a type as the **set of values** it contains: $\llbracket t \rrbracket \subseteq \mathcal{V}$

$$t_1 \leq t_2 \stackrel{\text{def}}{\iff} \llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket$$

$\llbracket \text{int} \rrbracket$	$= \mathbb{Z}$	$\llbracket t_1 \vee t_2 \rrbracket$	$= \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$
$\llbracket \text{bool} \rrbracket$	$= \{\text{true}, \text{false}\}$	$\llbracket t_1 \wedge t_2 \rrbracket$	$= \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$
$\llbracket \text{any} \rrbracket$	$= \mathcal{V}$	$\llbracket \neg t \rrbracket$	$= \mathcal{V} \setminus \llbracket t \rrbracket$
$\llbracket \text{empty} \rrbracket$	$= \emptyset$	$\llbracket t_1 \times t_2 \rrbracket$	$= \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$

What about arrows? The values of $t_1 \rightarrow t_2$ are λ -abstractions...

...whose typing depends on $\leq$: **circular definition!**

Semantic subtyping [Frisch et al., 2008]

Idea: interpret a type as the **set of values** it contains: $\llbracket t \rrbracket \subseteq \mathcal{V}$

$$t_1 \leq t_2 \stackrel{\text{def}}{\iff} \llbracket t_1 \rrbracket \subseteq \llbracket t_2 \rrbracket$$

$\llbracket \text{int} \rrbracket$	$= \mathbb{Z}$	$\llbracket t_1 \vee t_2 \rrbracket$	$= \llbracket t_1 \rrbracket \cup \llbracket t_2 \rrbracket$
$\llbracket \text{bool} \rrbracket$	$= \{\text{true}, \text{false}\}$	$\llbracket t_1 \wedge t_2 \rrbracket$	$= \llbracket t_1 \rrbracket \cap \llbracket t_2 \rrbracket$
$\llbracket \text{any} \rrbracket$	$= \mathcal{V}$	$\llbracket \neg t \rrbracket$	$= \mathcal{V} \setminus \llbracket t \rrbracket$
$\llbracket \text{empty} \rrbracket$	$= \emptyset$	$\llbracket t_1 \times t_2 \rrbracket$	$= \llbracket t_1 \rrbracket \times \llbracket t_2 \rrbracket$

What about arrows? The values of $t_1 \rightarrow t_2$ are λ -abstractions...

...whose typing depends on $\leq$: **circular definition!**

Solution: interpret a function as an input/output **relation**

$$\llbracket t_1 \rightarrow t_2 \rrbracket = \{ R \subseteq \mathcal{V} \times \mathcal{V} \mid \forall (v, v') \in R. v \in \llbracket t_1 \rrbracket \Rightarrow v' \in \llbracket t_2 \rrbracket \}^1$$

¹Simplified version: this definition is incorrect for cardinality reasons.

Exercise

Using the set-theoretic interpretation, justify that:

1. $\neg(\neg t) \simeq t$ (*double negation*)
2. $t \wedge (t_1 \vee t_2) \simeq (t \wedge t_1) \vee (t \wedge t_2)$ (*distributivity of $\wedge$ over $\vee$*)
3. $t \vee (t_1 \wedge t_2) \simeq (t \vee t_1) \wedge (t \vee t_2)$ (*distributivity of $\vee$ over $\wedge$*)
4. $(\text{int} \times \text{int}) \wedge \neg(\text{int} \times \text{bool}) \simeq \text{int} \times \text{int}$
5. $(t \times t_1) \wedge \neg(t \times t_2) \simeq t \times (t_1 \wedge \neg t_2)$

where $t_1 \simeq t_2$ means $t_1 \leq t_2$ and $t_2 \leq t_1$.

Subsumption rule:

$$[\text{Sub}] \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t'} \quad t \leq t'$$

Type system

Subsumption rule:

$$[\text{Sub}] \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t'} \quad t \leq t'$$

[Sub] is not **syntax-directed**: we do not know when to use it!

Solution: we integrate it into the application rule.

$$[\text{App}] \frac{\Gamma \vdash e_1 : s \rightarrow t \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : t} \quad \rightsquigarrow \quad [\text{App}] \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 e_2 : t} \quad t_1 \leq t_2 \rightarrow t$$

Type system

Subsumption rule:

$$[\text{Sub}] \frac{\Gamma \vdash e : t}{\Gamma \vdash e : t'} t \leq t'$$

[Sub] is not **syntax-directed**: we do not know when to use it!

Solution: we integrate it into the application rule.

$$[\text{App}] \frac{\Gamma \vdash e_1 : s \rightarrow t \quad \Gamma \vdash e_2 : s}{\Gamma \vdash e_1 e_2 : t} \quad \rightsquigarrow \quad [\text{App}] \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 e_2 : t} t_1 \leq t_2 \rightarrow t$$

Problem: we must **guess** t , knowing that the type t_1 of the function is not necessarily a simple type (e.g. $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$).

Syntax-directed application

We compute the **domain** of t_1 , and the **smallest possible result** t :

$$\text{dom}(t_1) = \max\{ s \mid t_1 \leq s \rightarrow \text{any} \} \qquad t_1 \circ t_2 = \min\{ t \mid t_1 \leq t_2 \rightarrow t \}$$

- $\text{dom}(t_1)$: the largest argument type accepted by the function
- $t_1 \circ t_2$: the smallest result type for an argument of type t_2

Syntax-directed application

We compute the **domain** of t_1 , and the **smallest possible result** t :

$$\text{dom}(t_1) = \max\{ s \mid t_1 \leq s \rightarrow \text{any} \} \qquad t_1 \circ t_2 = \min\{ t \mid t_1 \leq t_2 \rightarrow t \}$$

- $\text{dom}(t_1)$: the largest argument type accepted by the function
- $t_1 \circ t_2$: the smallest result type for an argument of type t_2

$$[\text{App}] \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \ e_2 : t_1 \circ t_2} \quad t_1 \leq \text{empty} \rightarrow \text{any} \quad t_2 \leq \text{dom}(t_1)$$

Syntax-directed application

We compute the **domain** of t_1 , and the **smallest possible result** t :

$$\text{dom}(t_1) = \max\{s \mid t_1 \leq s \rightarrow \text{any}\} \qquad t_1 \circ t_2 = \min\{t \mid t_1 \leq t_2 \rightarrow t\}$$

- $\text{dom}(t_1)$: the largest argument type accepted by the function
- $t_1 \circ t_2$: the smallest result type for an argument of type t_2

$$[\text{App}] \frac{\Gamma \vdash e_1 : t_1 \quad \Gamma \vdash e_2 : t_2}{\Gamma \vdash e_1 \ e_2 : t_1 \circ t_2} \quad t_1 \leq \text{empty} \rightarrow \text{any} \quad t_2 \leq \text{dom}(t_1)$$

Example: $t_1 = (\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$

- $\text{dom}(t_1) = \text{int} \vee \text{bool}$
- $t_1 \circ \text{int} = \text{int} \qquad t_1 \circ \text{bool} = \text{bool} \qquad t_1 \circ (\text{int} \vee \text{bool}) = \text{int} \vee \text{bool}$

Typecases, type narrowing and overloading

Different kinds of polymorphism

Type algebra and subtyping

Typecases, type narrowing and overloading

Gradual typing

Expressions $e ::= \dots \mid (e \in t) ? e : e$

- Tests **at runtime** whether the value of e has type t

Python: `isinstance(x, int)`, JavaScript: `typeof x === "number"`

Expressions $e ::= \dots \mid (e \in t) ? e : e$

- Tests **at runtime** whether the value of e has type t

Python: `isinstance(x, int)`, JavaScript: `typeof x === "number"`

- Semantics:

$$(v \in t) ? e_1 : e_2 \rightsquigarrow e_1 \quad \text{if } v \in \llbracket t \rrbracket$$
$$(v \in t) ? e_1 : e_2 \rightsquigarrow e_2 \quad \text{otherwise}$$

Expressions $e ::= \dots \mid (e \in t) ? e : e$

- Tests **at runtime** whether the value of e has type t

Python: `isinstance(x, int)`, JavaScript: `typeof x === "number"`

- Semantics:

$$(v \in t) ? e_1 : e_2 \rightsquigarrow e_1 \quad \text{if } v \in \llbracket t \rrbracket$$
$$(v \in t) ? e_1 : e_2 \rightsquigarrow e_2 \quad \text{otherwise}$$

- We can test whether an expression e is a function: $(e \in \text{empty} \rightarrow \text{any}) ? e_1 : e_2$

Python: `callable(x)`, JavaScript: `typeof x === "function"`

Expressions $e ::= \dots \mid (e \in t) ? e : e$

- Tests **at runtime** whether the value of e has type t

Python: `isinstance(x, int)`, JavaScript: `typeof x === "number"`

- Semantics:

$$(v \in t) ? e_1 : e_2 \rightsquigarrow e_1 \quad \text{if } v \in \llbracket t \rrbracket$$
$$(v \in t) ? e_1 : e_2 \rightsquigarrow e_2 \quad \text{otherwise}$$

- We can test whether an expression e is a function: $(e \in \text{empty} \rightarrow \text{any}) ? e_1 : e_2$

Python: `callable(x)`, JavaScript: `typeof x === "function"`

- ... but we generally cannot be more precise (e.g. test $e \in \text{int} \rightarrow \text{bool}$)
since the type of functions is not known at runtime!

Typing a typecase

First idea:

$$[\text{Case}] \frac{\Gamma \vdash e : t' \quad \Gamma \vdash e_1 : s \quad \Gamma \vdash e_2 : s}{\Gamma \vdash (e \in t) ? e_1 : e_2 : s}$$

Typing a typecase

First idea:

$$[\text{Case}] \frac{\Gamma \vdash e : t' \quad \Gamma \vdash e_1 : s \quad \Gamma \vdash e_2 : s}{\Gamma \vdash (e \in t) ? e_1 : e_2 : s}$$

```
def f(x):    # x : int | str
    if isinstance(x, int):
        return x + 1
    else:
        return len(x)
```

With [Case], $x : \text{int} \vee \text{string}$
in both branches...

Typing a typecase

First idea:

$$[\text{Case}] \frac{\Gamma \vdash e : t' \quad \Gamma \vdash e_1 : s \quad \Gamma \vdash e_2 : s}{\Gamma \vdash (e \in t) ? e_1 : e_2 : s}$$

```
def f(x):    # x : int | str
    if isinstance(x, int):
        return x + 1
    else:
        return len(x)
```

With [Case], $x : \text{int} \vee \text{string}$
in both branches...

$\Rightarrow x + 1$ and $\text{len}(x)$ are **ill-typed**!

Typing a typecase

First idea:

$$[\text{Case}] \frac{\Gamma \vdash e : t' \quad \Gamma \vdash e_1 : s \quad \Gamma \vdash e_2 : s}{\Gamma \vdash (e \in t) ? e_1 : e_2 : s}$$

```
def f(x):    # x : int | str
    if isinstance(x, int):
        return x + 1
    else:
        return len(x)
```

With [Case], $x : \text{int} \vee \text{string}$
in both branches...

$\Rightarrow x + 1$ and $\text{len}(x)$ are **ill-typed**!

We must **refine** the type of x in each branch,
using the information provided by the test.

Type narrowing (*occurrence typing*)

When the tested expression is a **variable**:

$$\text{[Case-Var]} \frac{\Gamma \vdash x : t' \quad \Gamma, x : t' \wedge t \vdash e_1 : s \quad \Gamma, x : t' \wedge \neg t \vdash e_2 : s}{\Gamma \vdash (x \in t) ? e_1 : e_2 : s}$$

Type narrowing (*occurrence typing*)

When the tested expression is a **variable**:

$$\text{[Case-Var]} \frac{\Gamma \vdash x : t' \quad \Gamma, x : t' \wedge t \vdash e_1 : s \quad \Gamma, x : t' \wedge \neg t \vdash e_2 : s}{\Gamma \vdash (x \in t) ? e_1 : e_2 : s}$$

On the example: $t' = \text{int} \vee \text{string}$ and $t = \text{int}$

- **then** branch: $(\text{int} \vee \text{string}) \wedge \text{int} \simeq \text{int}$
- **else** branch: $(\text{int} \vee \text{string}) \wedge \neg \text{int} \simeq \text{string}$

Type narrowing (*occurrence typing*)

When the tested expression is a **variable**:

$$\text{[Case-Var]} \frac{\Gamma \vdash x : t' \quad \Gamma, x : t' \wedge t \vdash e_1 : s \quad \Gamma, x : t' \wedge \neg t \vdash e_2 : s}{\Gamma \vdash (x \in t) ? e_1 : e_2 : s}$$

On the example: $t' = \text{int} \vee \text{string}$ and $t = \text{int}$

- **then** branch: $(\text{int} \vee \text{string}) \wedge \text{int} \simeq \text{int}$
- **else** branch: $(\text{int} \vee \text{string}) \wedge \neg \text{int} \simeq \text{string}$
- Note the use of **negation** $\neg t$ when the test fails!

Type narrowing (*occurrence typing*)

When the tested expression is a **variable**:

$$\text{[Case-Var]} \frac{\Gamma \vdash x : t' \quad \Gamma, x : t' \wedge t \vdash e_1 : s \quad \Gamma, x : t' \wedge \neg t \vdash e_2 : s}{\Gamma \vdash (x \in t) ? e_1 : e_2 : s}$$

On the example: $t' = \text{int} \vee \text{string}$ and $t = \text{int}$

- **then** branch: $(\text{int} \vee \text{string}) \wedge \text{int} \simeq \text{int}$
- **else** branch: $(\text{int} \vee \text{string}) \wedge \neg \text{int} \simeq \text{string}$
- Note the use of **negation** $\neg t$ when the test fails!
- If $t' \wedge t \simeq \text{empty}$ (resp. $t' \wedge \neg t \simeq \text{empty}$), the branch cannot be executed (**dead code**): we do not type it (or we emit a warning)

Narrowing: beyond variables

In practice, more complex expressions are also tested:

```
if isinstance(p[0], int): ...  
if x is not None and x > 0: ...  
is_int = isinstance(x, int)  
if is_int: ...                # alias of the test
```

Narrowing: beyond variables

In practice, more complex expressions are also tested:

```
if isinstance(p[0], int): ...  
if x is not None and x > 0: ...  
is_int = isinstance(x, int)  
if is_int: ...                # alias of the test
```

- Introduced by Typed Racket [Tobin-Hochstadt and Felleisen, 2008], generalized to arbitrary expressions with set-theoretic types [Castagna et al., 2022]
- TypeScript, Pyright, ... support part of it (*type guards*, *type narrowing*)

Overloaded functions

With typecases, we can write **overloaded** functions (cf. `neg` in Python):

$$\text{neg} \equiv \lambda x. (x \in \text{bool}) ? \text{not } x : \text{minus } x$$

(with `not` : `bool` $\rightarrow$ `bool` and `minus` : `int` $\rightarrow$ `int`)

Overloaded functions

With typecases, we can write **overloaded** functions (cf. `neg` in Python):

$$\text{neg} \equiv \lambda x. (x \in \text{bool}) ? \text{not } x : \text{minus } x$$

(with `not` : `bool` $\rightarrow$ `bool` and `minus` : `int` $\rightarrow$ `int`)

Usual abstraction ($x : \text{int} \vee \text{bool}$): `neg` : (`int` $\vee$ `bool`) $\rightarrow$ (`int` $\vee$ `bool`)

Correct, but **imprecise**: the link between input and output is lost.

Overloaded functions

With typecases, we can write **overloaded** functions (cf. `neg` in Python):

$$\text{neg} \equiv \lambda x. (x \in \text{bool}) ? \text{not } x : \text{minus } x$$

(with `not` : `bool` $\rightarrow$ `bool` and `minus` : `int` $\rightarrow$ `int`)

Usual abstraction ($x : \text{int} \vee \text{bool}$): `neg` : (`int` $\vee$ `bool`) $\rightarrow$ (`int` $\vee$ `bool`)

Correct, but **imprecise**: the link between input and output is lost.

$$\text{neg} : (\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$$

Overloaded functions

With typecases, we can write **overloaded** functions (cf. `neg` in Python):

$$\text{neg} \equiv \lambda x. (x \in \text{bool}) ? \text{not } x : \text{minus } x$$

(with `not` : `bool` $\rightarrow$ `bool` and `minus` : `int` $\rightarrow$ `int`)

Usual abstraction ($x : \text{int} \vee \text{bool}$): `neg` : (`int` $\vee$ `bool`) $\rightarrow$ (`int` $\vee$ `bool`)

Correct, but **imprecise**: the link between input and output is lost.

$$\text{neg} : (\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$$

Idea: type the function **several times**, once per case, then combine:

$$[\text{Inter}] \frac{\Gamma \vdash e : t_1 \quad \Gamma \vdash e : t_2}{\Gamma \vdash e : t_1 \wedge t_2}$$

With `neg` $\equiv \lambda x. e$ and $e \equiv (x \in \text{bool}) ? \text{not } x : \text{minus } x$:

$$\vdash \text{neg} : (\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$$

With `neg` $\equiv \lambda x. e$ and $e \equiv (x \in \text{bool}) ? \text{not } x : \text{minus } x$:

$$[\text{Inter}] \frac{\vdash \text{neg} : \text{int} \rightarrow \text{int} \quad \vdash \text{neg} : \text{bool} \rightarrow \text{bool}}{\vdash \text{neg} : (\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})}$$

With `neg` $\equiv \lambda x. e$ and $e \equiv (x \in \text{bool}) ? \text{not } x : \text{minus } x$:

$$\text{[Inter]} \frac{\begin{array}{c} \text{[Abs]} \frac{x : \text{int} \vdash e : \text{int}}{\vdash \text{neg} : \text{int} \rightarrow \text{int}} \quad \text{[Abs]} \frac{x : \text{bool} \vdash e : \text{bool}}{\vdash \text{neg} : \text{bool} \rightarrow \text{bool}} \end{array}}{\vdash \text{neg} : (\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})}$$

Typing `neg`

With `neg` $\equiv \lambda x. e$ and $e \equiv (x \in \text{bool}) ? \text{not } x : \text{minus } x$:

$$\begin{array}{c} \text{[Case-Var]} \frac{x : \text{int} \vdash \text{minus } x : \text{int}}{x : \text{int} \vdash e : \text{int}} \quad \text{[Case-Var]} \frac{x : \text{bool} \vdash \text{not } x : \text{bool}}{x : \text{bool} \vdash e : \text{bool}} \\ \text{[Abs]} \frac{}{\vdash \text{neg} : \text{int} \rightarrow \text{int}} \quad \text{[Abs]} \frac{}{\vdash \text{neg} : \text{bool} \rightarrow \text{bool}} \\ \text{[Inter]} \frac{}{\vdash \text{neg} : (\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})} \end{array}$$

With `neg` $\equiv \lambda x. e$ and `e` $\equiv (x \in \text{bool}) ? \text{not } x : \text{minus } x$:

$$\begin{array}{c}
 \text{[Case-Var]} \frac{x : \text{int} \vdash \text{minus } x : \text{int}}{x : \text{int} \vdash e : \text{int}} \quad \text{[Case-Var]} \frac{x : \text{bool} \vdash \text{not } x : \text{bool}}{x : \text{bool} \vdash e : \text{bool}} \\
 \text{[Abs]} \frac{}{\vdash \text{neg} : \text{int} \rightarrow \text{int}} \quad \text{[Abs]} \frac{}{\vdash \text{neg} : \text{bool} \rightarrow \text{bool}} \\
 \text{[Inter]} \frac{}{\vdash \text{neg} : (\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})}
 \end{array}$$

- Left: $\text{int} \wedge \text{bool} \simeq \text{empty}$, the `then` branch is ignored
- Right: $\text{bool} \wedge \neg \text{bool} \simeq \text{empty}$, the `else` branch is ignored

Overloading in practice

- `[Inter]` is not syntax-directed:
how many times should the function be typed, and with which types for x ?

Overloading in practice

- [Inter] is not syntax-directed:
how many times should the function be typed, and with which types for x ?
- Common solution: the programmer **annotates** the function

$$[\text{Abs-Inter}] \frac{\forall i \in I. \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\bigwedge_{i \in I} s_i \rightarrow t_i} x. e : \bigwedge_{i \in I} s_i \rightarrow t_i}$$

e.g. overloaded signatures in TypeScript, `@overload` in Python

Overloading in practice

- [Inter] is not syntax-directed:
how many times should the function be typed, and with which types for x ?
- Common solution: the programmer **annotates** the function

$$[\text{Abs-Inter}] \frac{\forall i \in I. \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\wedge_{i \in I} s_i \rightarrow t_i} x. e : \wedge_{i \in I} s_i \rightarrow t_i}$$

e.g. overloaded signatures in TypeScript, `@overload` in Python

- Application reuses the $\circ$ operator seen previously:

`neg 3 : ((int → int) ∧ (bool → bool)) ∘ int = int`

Overloading in practice

- [Inter] is not syntax-directed:
how many times should the function be typed, and with which types for x ?
- Common solution: the programmer **annotates** the function

$$[\text{Abs-Inter}] \frac{\forall i \in I. \Gamma, x : s_i \vdash e : t_i}{\Gamma \vdash \lambda^{\bigwedge_{i \in I} s_i \rightarrow t_i} x. e : \bigwedge_{i \in I} s_i \rightarrow t_i}$$

e.g. overloaded signatures in TypeScript, `@overload` in Python

- Application reuses the $\circ$ operator seen previously:

`neg 3 : ((int  $\rightarrow$  int)  $\wedge$  (bool  $\rightarrow$  bool))  $\circ$  int = int`

- **Inference** of intersection types is possible in some cases [Castagna et al., 2024]

Exercise

1. With `neg` : $(\text{int} \rightarrow \text{int}) \wedge (\text{bool} \rightarrow \text{bool})$ and $x : \text{int} \vee \text{bool}$, what is the type of `neg x`?
2. Assume `parse` : $\text{string} \rightarrow \text{int}$. Give two types for

$$\text{to_int} \equiv \lambda x. (x \in \text{int}) ? x : ((x \in \text{string}) ? \text{parse } x : 0)$$

one with intersections, the other without. Are they equivalent?

3. Why can't we type `neg` with $(\text{int} \vee \text{bool}) \rightarrow (\text{int} \vee \text{bool})$ only, if we want to accept `(neg 3) + 1`?

Gradual typing

Different kinds of polymorphism

Type algebra and subtyping

Typecases, type narrowing and overloading

Gradual typing

Statically unknown types

```
user = json.loads(open("user.json").read())  
print(user["name"].upper())  
age = user["age"] + 1
```

Which type for `user`?

Statically unknown types

```
user = json.loads(open("user.json").read())  
print(user["name"].upper())  
age = user["age"] + 1
```

Which type for `user`? It depends on the **content of the file**, only known at runtime!

Statically unknown types

```
user = json.loads(open("user.json").read())  
print(user["name"].upper())  
age = user["age"] + 1
```

Which type for `user`? It depends on the **content of the file**, only known at runtime!

- `user : any?` Then `user["name"]` is **rejected**
⇒ we would have to test the type of each field before using it (tedious)

Statically unknown types

```
user = json.loads(open("user.json").read())  
print(user["name"].upper())  
age = user["age"] + 1
```

Which type for `user`? It depends on the **content of the file**, only known at runtime!

- `user : any`? Then `user["name"]` is **rejected**
⇒ we would have to test the type of each field before using it (tedious)
- `user : {name : string, age : int}`? Nothing guarantees it
⇒ the type checker **lies** if the file does not match

Statically unknown types

```
user = json.loads(open("user.json").read())  
print(user["name"].upper())  
age = user["age"] + 1
```

Which type for `user`? It depends on the **content of the file**, only known at runtime!

- `user : any`? Then `user["name"]` is **rejected**
⇒ we would have to test the type of each field before using it (tedious)
- `user : {name : string, age : int}`? Nothing guarantees it
⇒ the type checker **lies** if the file does not match

We need a type meaning “**statically unknown, checked at runtime**”

⇒ the **dynamic type** ? [Siek and Taha, 2006]

(this is TypeScript’s `any` and Python’s `Any`, not to be confused with our `any`!)

The dynamic type

Gradual types $\tau ::= ? \mid b \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \dots$

The dynamic type

Gradual types $\tau ::= ? \mid b \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \dots$

- The type $?$ can be given to **any expression**
 $\Rightarrow ?$ sometimes behaves like any

The dynamic type

Gradual types $\tau ::= ? \mid b \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \dots$

- The type $?$ can be given to **any expression**
 $\Rightarrow ?$ sometimes behaves like `any`
- An expression of type $?$ can be used **anywhere**
 $\Rightarrow ?$ sometimes behaves like `empty`

The dynamic type

Gradual types $\tau ::= ? \mid b \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \dots$

- The type $?$ can be given to **any expression**
 $\Rightarrow ?$ sometimes behaves like `any`
- An expression of type $?$ can be used **anywhere**
 $\Rightarrow ?$ sometimes behaves like `empty`
- But $?$ is **not** a subtype and supertype of everything: `else  $\text{int} \leq ? \leq \text{bool}$`

The dynamic type

Gradual types $\tau ::= ? \mid b \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \dots$

- The type $?$ can be given to **any expression**
 $\Rightarrow ?$ sometimes behaves like `any`
- An expression of type $?$ can be used **anywhere**
 $\Rightarrow ?$ sometimes behaves like `empty`
- But $?$ is **not** a subtype and supertype of everything: else $\text{int} \leq ? \leq \text{bool}$
- Instead: a **consistency** relation $\tau_1 \sim \tau_2$, **not transitive**

$$\begin{array}{ccccc} \frac{}{? \sim \tau} & \frac{}{\tau \sim ?} & \frac{}{b \sim b} & \frac{\tau_1 \sim \tau'_1 \quad \tau_2 \sim \tau'_2}{\tau_1 \rightarrow \tau_2 \sim \tau'_1 \rightarrow \tau'_2} & \frac{\tau_1 \sim \tau'_1 \quad \tau_2 \sim \tau'_2}{\tau_1 \times \tau_2 \sim \tau'_1 \times \tau'_2} \end{array}$$

Examples: $\text{int} \rightarrow ? \sim ? \rightarrow \text{bool}$, $\text{int} \sim ? \sim \text{bool}$ but $\text{int} \not\sim \text{bool}$

Gradual typing [Siek and Taha, 2006]

We replace equality with consistency in the application rule:

$$[\text{App}] \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \tau_2 \sim \tau_1$$

$$[\text{App?}] \frac{\Gamma \vdash e_1 : ? \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : ?}$$

Gradual typing [Siek and Taha, 2006]

We replace equality with consistency in the application rule:

$$\begin{array}{c} \text{[App]} \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \tau_2 \sim \tau_1 \end{array} \quad \begin{array}{c} \text{[App?]} \frac{\Gamma \vdash e_1 : ? \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : ?} \end{array}$$

Exercise: derive the judgement $x : \text{int} \rightarrow ?$, $y : ? \vdash xyy : ?$

Gradual typing [Siek and Taha, 2006]

We replace equality with consistency in the application rule:

$$\begin{array}{c} \text{[App]} \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \tau_2 \sim \tau_1 \qquad \text{[App?]} \frac{\Gamma \vdash e_1 : ? \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : ?} \end{array}$$

Exercise: derive the judgement $x : \text{int} \rightarrow ?, y : ? \vdash xyy : ?$

- Program without $?$: classical **static** typing
- Program where everything is $?$: **dynamic** typing (everything is accepted)
- In between: annotations can be added **progressively**
(e.g. migrating an existing Python or JavaScript codebase)

Gradual typing [Siek and Taha, 2006]

We replace equality with consistency in the application rule:

$$\begin{array}{c} \text{[App]} \frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau} \tau_2 \sim \tau_1 \qquad \text{[App?]} \frac{\Gamma \vdash e_1 : ? \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : ?} \end{array}$$

Exercise: derive the judgement $x : \text{int} \rightarrow ?$, $y : ? \vdash xyy : ?$

- Program without $?$: classical **static** typing
- Program where everything is $?$: **dynamic** typing (everything is accepted)
- In between: annotations can be added **progressively**
(e.g. migrating an existing Python or JavaScript codebase)
- The type checker accepts programs that may fail...
 $\Rightarrow$ what could not be checked statically must be **checked at runtime**

Compilation to a language with *casts*

We insert **casts** $\langle \tau \Leftarrow \tau' \rangle e$ wherever consistency was used. For instance, if $y : ?$:

$$(\lambda x:\text{int}. x + 1) y \quad \rightsquigarrow \quad (\lambda x:\text{int}. x + 1) \langle \text{int} \Leftarrow ? \rangle y$$

Compilation to a language with *casts*

We insert **casts** $\langle \tau \Leftarrow \tau' \rangle e$ wherever consistency was used. For instance, if $y : ?$:

$$(\lambda x:\text{int}. x + 1) y \quad \rightsquigarrow \quad (\lambda x:\text{int}. x + 1) \langle \text{int} \Leftarrow ? \rangle y$$

Casts check types **at runtime**, stopping execution in case of error:

- Base types: immediate check

$$\langle \text{int} \Leftarrow ? \rangle 3 \rightsquigarrow 3 \qquad \langle \text{int} \Leftarrow ? \rangle \text{true} \rightsquigarrow \text{error}$$

Compilation to a language with *casts*

We insert **casts** $\langle \tau \Leftarrow \tau' \rangle e$ wherever consistency was used. For instance, if $y : ?$:

$$(\lambda x:\text{int}. x + 1) y \quad \rightsquigarrow \quad (\lambda x:\text{int}. x + 1) \langle \text{int} \Leftarrow ? \rangle y$$

Casts check types **at runtime**, stopping execution in case of error:

- Base types: immediate check

$$\langle \text{int} \Leftarrow ? \rangle 3 \rightsquigarrow 3 \qquad \langle \text{int} \Leftarrow ? \rangle \text{true} \rightsquigarrow \mathbf{error}$$

- Functions: impossible to check right away (cf. typecases)
 $\Rightarrow$ we **wrap** the function, and check the argument and the result at each call

Compilation to a language with *casts*

We insert **casts** $\langle \tau \Leftarrow \tau' \rangle e$ wherever consistency was used. For instance, if $y : ?$:

$$(\lambda x:\text{int}. x + 1) y \quad \rightsquigarrow \quad (\lambda x:\text{int}. x + 1) \langle \text{int} \Leftarrow ? \rangle y$$

Casts check types **at runtime**, stopping execution in case of error:

- Base types: immediate check

$$\langle \text{int} \Leftarrow ? \rangle 3 \rightsquigarrow 3 \qquad \langle \text{int} \Leftarrow ? \rangle \text{true} \rightsquigarrow \mathbf{error}$$

- Functions: impossible to check right away (cf. typecases)
 $\Rightarrow$ we **wrap** the function, and check the argument and the result at each call
- **Blame**: on failure, we point to the responsible cast (hence part of the code)
“well-typed programs can’t be blamed” (Wadler & Findler)

Gradual typing in practice

TypeScript, MyPy, Pyright: **optional typing**, types are **erased** at runtime

```
const user: any = JSON.parse(s);  
const age: number = user.age;    // no check at all!  
age + 1;                          // unexpected behaviour if age is a string
```

Gradual typing in practice

TypeScript, MyPy, Pyright: **optional typing**, types are **erased** at runtime

```
const user: any = JSON.parse(s);  
const age: number = user.age;    // no check at all!  
age + 1;                        // unexpected behaviour if age is a string
```

- No casts $\Rightarrow$ **unsound**: a value of type **number** may be a **string**!

Gradual typing in practice

TypeScript, MyPy, Pyright: **optional typing**, types are **erased** at runtime

```
const user: any = JSON.parse(s);  
const age: number = user.age;    // no check at all!  
age + 1;                        // unexpected behaviour if age is a string
```

- No casts $\Rightarrow$ **unsound**: a value of type **number** may be a **string**!
- **Sound** gradual typing: Typed Racket, Safe TypeScript [Rastogi et al., 2015], ... but casts (especially on functions) have a runtime **cost** [Greenberg, 2019]

Gradual typing in practice

TypeScript, MyPy, Pyright: **optional typing**, types are **erased** at runtime

```
const user: any = JSON.parse(s);  
const age: number = user.age;    // no check at all!  
age + 1;                        // unexpected behaviour if age is a string
```

- No casts $\Rightarrow$ **unsound**: a value of type **number** may be a **string**!
- **Sound** gradual typing: Typed Racket, Safe TypeScript [Rastogi et al., 2015], ... but casts (especially on functions) have a runtime **cost** [Greenberg, 2019]
- Trade-off: **shallow** checks (*transient*): only the head constructor is checked (“is it a function?”), not its full type

Gradual typing and set-theoretic types

Gradual types $\tau ::= ? \mid b \mid \alpha \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg t \mid \dots$

- ? represents an **unknown static** type

Gradual typing and set-theoretic types

Gradual types $\tau ::= ? \mid b \mid \alpha \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg t \mid \dots$

- $?$ represents an **unknown static** type
- $\text{int} \vee ?$: “at least int”, $\text{int} \wedge ?$: “an int, maybe more precise”

Gradual typing and set-theoretic types

Gradual types $\tau ::= ? \mid b \mid \alpha \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg t \mid \dots$

- $?$ represents an **unknown static** type
- $\text{int} \vee ?$: “at least int”, $\text{int} \wedge ?$: “an int, maybe more precise”
- **Materialization**: $\tau \leq \tau'$ if τ' is obtained by replacing occurrences of $?$ in τ with types (e.g. $? \rightarrow \text{int} \leq \text{bool} \rightarrow \text{int}$)

Gradual typing and set-theoretic types

Gradual types $\tau ::= ? \mid b \mid \alpha \mid \tau \rightarrow \tau \mid \tau \times \tau \mid \tau \vee \tau \mid \tau \wedge \tau \mid \neg t \mid \dots$

- $?$ represents an **unknown static** type
- $\text{int} \vee ?$: “at least `int`”, $\text{int} \wedge ?$: “an `int`, maybe more precise”
- **Materialization**: $\tau \leq \tau'$ if τ' is obtained by replacing occurrences of $?$ in τ with types (e.g. $? \rightarrow \text{int} \leq \text{bool} \rightarrow \text{int}$)
- The system of [Castagna et al., 2019] combines subtyping and materialization:

$$[\text{Sub}] \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \tau \leq \tau'$$

$$[\text{Mat}] \frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \tau \leq \tau'$$

Back to the example

```
user = json.loads(s)                                # user : ?
if isinstance(user["age"], int):
    age = user["age"] + 1                            # user["age"] : ? & int
else:
    age = 0
```

How many casts are needed for sound typing?

Back to the example

```
user = json.loads(s)                                # user : ?
if isinstance(user["age"], int):
    age = user["age"] + 1                            # user["age"] : ? & int
else:
    age = 0
```

How many casts are needed for sound typing?

- `user["age"]` requires `user` to be a dictionary, but `user` has type ?
⇒ a cast must be inserted

Back to the example

```
user = json.loads(s)                                # user : ?
if isinstance(user["age"], int):
    age = user["age"] + 1                            # user["age"] : ? & int
else:
    age = 0
```

How many casts are needed for sound typing?

- `user["age"]` requires `user` to be a dictionary, but `user` has type ?
⇒ a cast must be inserted
- The typecase narrows the gradual type: in the `then` branch, `user["age"]` has type $? \wedge \text{int}$ (all its materializations are subtypes of `int`)
⇒ `+ 1` is **safe**: no cast needed

Back to the example

```
user = json.loads(s)                                # user : ?
if isinstance(user["age"], int):
    age = user["age"] + 1                            # user["age"] : ? & int
else:
    age = 0
```

How many casts are needed for sound typing?

- `user["age"]` requires `user` to be a dictionary, but `user` has type ?
⇒ a cast must be inserted
- The typecase narrows the gradual type: in the `then` branch, `user["age"]` has type $? \wedge \text{int}$ (all its materializations are subtypes of `int`)
⇒ `+ 1` is **safe**: no cast needed

Typecases, narrowing and gradual typing combine naturally!

Summary

- Dynamic languages rely on idioms that are hard to type:
heterogeneous data, type tests, overloading, dynamically loaded data

Summary

- Dynamic languages rely on idioms that are hard to type:
heterogeneous data, type tests, overloading, dynamically loaded data
- **Set-theoretic types**: unions, intersections, negations (and type variables)
 - **Unions**: heterogeneous values, **intersections**: overloading (among others)

Summary

- Dynamic languages rely on idioms that are hard to type:
heterogeneous data, type tests, overloading, dynamically loaded data
- **Set-theoretic types**: unions, intersections, negations (and type variables)
 - **Unions**: heterogeneous values, **intersections**: overloading (among others)
- **Algebraic subtyping**: defined by inference rules, simple but limited

Summary

- Dynamic languages rely on idioms that are hard to type:
heterogeneous data, type tests, overloading, dynamically loaded data
- **Set-theoretic types**: unions, intersections, negations (and type variables)
 - **Unions**: heterogeneous values, **intersections**: overloading (among others)
- **Algebraic subtyping**: defined by inference rules, simple but limited
- **Semantic subtyping**: defined by inclusion of sets of values, decidable

Summary

- Dynamic languages rely on idioms that are hard to type:
heterogeneous data, type tests, overloading, dynamically loaded data
- **Set-theoretic types**: unions, intersections, negations (and type variables)
 - **Unions**: heterogeneous values, **intersections**: overloading (among others)
- **Algebraic subtyping**: defined by inference rules, simple but limited
- **Semantic subtyping**: defined by inclusion of sets of values, decidable
- **Typecases** and **narrowing**: refine types, type overloading

Summary

- Dynamic languages rely on idioms that are hard to type:
heterogeneous data, type tests, overloading, dynamically loaded data
- **Set-theoretic types**: unions, intersections, negations (and type variables)
 - **Unions**: heterogeneous values, **intersections**: overloading (among others)
- **Algebraic subtyping**: defined by inference rules, simple but limited
- **Semantic subtyping**: defined by inclusion of sets of values, decidable
- **Typecases** and **narrowing**: refine types, type overloading
- **Gradual typing**: the type ? for statically unknown information,
checked at runtime by casts

Summary

- Dynamic languages rely on idioms that are hard to type:
heterogeneous data, type tests, overloading, dynamically loaded data
- **Set-theoretic types**: unions, intersections, negations (and type variables)
 - **Unions**: heterogeneous values, **intersections**: overloading (among others)
- **Algebraic subtyping**: defined by inference rules, simple but limited
- **Semantic subtyping**: defined by inclusion of sets of values, decidable
- **Typecases** and **narrowing**: refine types, type overloading
- **Gradual typing**: the type ? for statically unknown information, checked at runtime by casts
- These ideas are used in TypeScript, Pyright, Luau, Elixir, ...



Castagna, G., Duboc, G., and Valim, J. (2023).

The design principles of the elixir type system.

The Art, Science, and Engineering of Programming, 8(2).



Castagna, G., Lanvin, V., Petrucciani, T., and Siek, J. G. (2019).

Gradual typing: a new perspective.

Proc. ACM Program. Lang., 3(POPL).



Castagna, G., Laurent, M., and Nguyễn, K. (2024).

Polymorphic type inference for dynamic languages.

Proc. ACM Program. Lang., 8(POPL).

-  Castagna, G., Laurent, M., Nguyen, K., and Lutze, M. (2022).
On type-cases, union elimination, and occurrence typing.
Proc. ACM Program. Lang., 6(POPL).
-  Frisch, A., Castagna, G., and Benzaken, V. (2008).
Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types.
Journal of the ACM, 55(4):19:1–19:64.
-  Greenberg, M. (2019).
The Dynamic Practice and Static Theory of Gradual Typing.
In *3rd Summit on Advances in Programming Languages (SNAPL 2019)*, volume 136 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 6:1–6:20.



Jeffrey, A. (2022).

Semantic subtyping in luau.

Roblox Technical Blog.

Accessed on May 6th 2023.



Laurent, M. and Vitek, J. (2026).

Type inference for functional and imperative dynamic languages .

working paper or preprint.

-  Rastogi, A., Swamy, N., Fournet, C., Bierman, G., and Vekris, P. (2015).
Safe and efficient gradual typing for typescript.
In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, page 167–180, New York, NY, USA. Association for Computing Machinery.
-  Schimpf, A., Wehr, S., and Bieniusa, A. (2023).
Set-theoretic types for erlang.
In *Proceedings of the 34th Symposium on Implementation and Application of Functional Languages*, IFL '22, New York, NY, USA. Association for Computing Machinery.

-  Siek, J. G. and Taha, W. (2006).
Gradual typing for functional languages.
In *Scheme and Functional Programming Workshop*, pages 81–92.
-  Tobin-Hochstadt, S. and Felleisen, M. (2008).
The design and implementation of typed scheme.
In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 395–406, New York, NY, USA. ACM.