Records
We'll use records to motivate subtyping:
\[
e ::= \cdots | \{ l_1 = e_1, \ldots, l_n = e_n \} | e.l
\]
\[
\tau ::= \cdots | \{ l_1 : \tau_1, \ldots, l_n : \tau_n \}
\]
\[
\Gamma \vdash e : \{ l_1 = \tau_1, \ldots, l_n = \tau_n \} \quad 1 \leq i \leq n
\]
\[
\Gamma \vdash e.l : \tau
\]
Should this typecheck?
\[
(\lambda x : \{ l_1 : \text{int}, l_2 : \text{int} \}. x.l_1 + x.l_2) \{ l_1 = 3, l_2 = 4, l_3 = 5 \}
\]
Right now, it doesn't, but it won't get stuck
Suggests width subtyping:
\[
\tau_1 \leq \tau_2
\]
And one new type-checking rule: Subsumption
\[
\frac{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}{\Gamma \vdash e : \tau}
\]
Now it type-checks

\[
\begin{align*}
\Gamma & : (x_1 : \text{int}, x_2 : \text{int}) \vdash x_1 + x_2 : \text{int} \\
\vdash (\lambda x : \{l_1 : \text{int}, l_2 : \text{int}\}.x.l_1 + x.l_2) : \{l_1 : \text{int}, l_2 : \text{int}\} \rightarrow \text{int}
\end{align*}
\]

Instantiation of Subsumption is highlighted (pardon formatting)

The derivation of the subtyping fact
\{l_1: \text{int}, l_2: \text{int}, l_3: \text{int}\} \leq \{l_1: \text{int}, l_2: \text{int}\}
would continue, using rules for the \(\tau_1 \leq \tau_2\) judgment
▶ But here we just use the one axiom we have so far

Clean division of responsibility:
▶ Where to use subsumption
▶ How to show two types are subtypes

Digression: Efficiency

With our semantics, width and permutation subtyping make perfect sense

But it would be nice to compile \(e.l\) down to:
1. evaluate \(e\) to a record stored at an address \(a\)
2. load \(a\) into a register \(r_1\)
3. load field \(l\) from a fixed offset (e.g., 4) into \(r_2\)

Many type systems are engineered to make this easy for compiler writers

Makes restrictions seem odd if you do not know techniques for implementing high-level languages

Permutation

Does this program type-check? Does it get stuck?

\(\lambda x: \{l_1: \text{int}, l_2: \text{int}\}.x.l_1 + x.l_2\)\{l_2=3; l_1=4\}

Suggests permutation subtyping:

\[\{l_1: \tau_1, \ldots, l_{i-1}: \tau_{i-1}, l_i: \tau_i, \ldots, l_n: \tau_n\} \leq \{l_1: \tau_1, \ldots, l_i: \tau_i, l_{i-1}: \tau_{i-1}, \ldots, l_n: \tau_n\}\]

Example with width and permutation: Show
\(\vdash \{l_1=7, l_2=8, l_3=9\} : \{l_2: \text{int}, l_1: \text{int}\}\)

It’s no longer clear there is an (efficient, sound, complete) type-checking algorithm
▶ They sometimes exist and sometimes don’t
▶ Here they do

The Top type

It is convenient to have a type that is a supertype of every type.

\[S \leq \text{Top}\]

\(\text{Top}\) is a new type constant that defines the maximum element of the subtype relation.

It corresponds to the type \text{Object} found in many OO languages.

Digression continued

With width subtyping alone, the strategy is easy

With permutation subtyping alone, it’s easy but have to “alphabetize”

With both, it’s not easy...

\[
\begin{align*}
f_1 : \{l_1 : \text{int}\} \rightarrow \text{int} & \quad f_2 : \{l_2 : \text{int}\} \rightarrow \text{int} \\
x_1 = \{l_1 = 0, l_2 = 0\} & \quad x_2 = \{l_2 = 0, l_3 = 0\} \\
f_1(x_1) & \quad f_2(x_1) \quad f_2(x_2)
\end{align*}
\]

Can use dictionary-passing (look up offset at run-time) and maybe optimize away (some) lookups

Named types can avoid this, but make code less flexible
So far

- A new subtyping judgement and a new typing rule
- Width, permutation, and transitivity

$$\tau_1 \leq \tau_2$$

$$\{l_1 \vdash \tau_1, \ldots, l_n \vdash \tau_n, l : \tau\} \leq \{l_1 \vdash \tau_1, \ldots, l_n \vdash \tau_n\}$$

$$\begin{array}{c}
\{l_1 \vdash \tau_1, \ldots, l_{i-1} \vdash \tau_{i-1}, l_i \vdash \tau_i, l_{i+1} \vdash \tau_{i+1}, \ldots, l_n \vdash \tau_n\} \\
\{l_1 \vdash \tau_1, \ldots, l_{i-1} \vdash \tau_{i-1}, l_i \vdash \tau_i, l_{i+1} \vdash \tau_{i+1}, \ldots, l_n \vdash \tau_n\}
\end{array}

\frac{\tau_1 \leq \tau_2}{\tau_1 \leq \tau_2 \leq \tau_3}

\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \tau'} \frac{\tau' \leq \tau}{\Gamma \vdash e : \tau}

Now: This is all much more useful if we extend subtyping so it can be used on “parts” of larger types:
- Example: Can’t yet use subsumption on a record field’s type
- Example: There are no supertypes yet of $\tau_1 \rightarrow \tau_2$

Function subtyping

Given our rich subtyping on records (and/or other primitives), how do we extend it to other types, notably $\tau_1 \rightarrow \tau_2$?

For example, we’d like $\text{int} \rightarrow \{l_1 : \text{int}, l_2 : \text{int}\} \leq \text{int} \rightarrow \{l_1 : \text{int}\}$ so we can pass a function of the subtype somewhere expecting a function of the supertype

$$\begin{array}{c}
??? \\
\tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4
\end{array}

For a function to have type $\tau_3 \rightarrow \tau_4$ it must return something of type $\tau_4$ (including subtypes) whenever given something of type $\tau_3$ (including subtypes). A function assuming less than $\tau_3$ will do, but not one assuming more. A function returning more than $\tau_4$ but not one returning less.

Covariance and Contravariance

Given types $S$ and $T$ such that $S \leq T$ (also written as $S <: T$):
- **Covariant**: $S$ and $T$ are said to be covariant when the more specific type, $S$, can be used when a more generic type, $T$, is specified. This applies to functions, i.e., a function that returns $S$ can be used in the same context as a function that returns $T$.

- **Contravariant**: $S$ and $T$ are contra variant when the more generic type, $T$, can be used where the more specific type, $S$, is specified. A function that takes an argument of type $T$ can be used in the same context as a function that takes an argument of type $S$.

- **Invariant**: The type specified is the only one that can be used.

Covariant Example (C++)

```cpp
class X {};
class Y : public X {};
class Z : public Y {};

class A {
public:
    virtual Y *foo() { return new Y(); }
};

class B : public A {
public:
    virtual Z *foo() { return new Z(); }
};
```

Here we have three classes $X$, $Y$, and $Z$ which we will return from a virtual function in classes $A$ and $B$. This code is valid because $B::$foo is returning a more specialized type than $A::$foo because $Z$ is a subtype of $Y$.

Depth

Does this program type-check? Does it get stuck?

$$(\lambda x : \{l_1 : \text{int}, l_2 : \text{int}\}. x.l_1.l_3 + x.l_2 \{l_1 = l_3 = 3, l_4 = 9\}, l_2 = 4)$$

Suggests depth subtyping

$$\begin{array}{c}
\{l_1 \vdash \tau_1, \ldots, l_i \vdash \tau_i, \ldots, l_n \vdash \tau_n\} \leq \{l_1 \vdash \tau_1, \ldots, l_i' \vdash \tau_i', \ldots, l_n \vdash \tau_n\}
\end{array}

(With permutation subtyping, can just have depth on left-most field)

Soundness of this rule depends crucially on fields being immutable!
- Depth subtyping is **unsound** in the presence of mutation
- Trade-off between power (mutation) and sound expressiveness (depth subtyping)

Function subtyping, cont’d

$$\frac{\tau_3 \leq \tau_1}{\tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4}$$

Also want: $\tau \leq \tau$

Example: $\lambda x : \{l_1 : \text{int}, l_2 : \text{int}\}. \{\text{int} = x.l_2, l_2 = x.l_1\}$

can have type $\{l_1 : \text{int}, l_2 : \text{int}, l_3 : \text{int}\} \rightarrow \{l_1 : \text{int}\}$

but not $\{l_1 : \text{int}\} \rightarrow \{l_1 : \text{int}\}$

Jargon: Function types are **contravariant** in their argument and **covariant** in their result
- Depth subtyping means immutable records are covariant in their fields

This is unintuitive enough that you, a friend, or a manager, will some day be convinced that functions can be covariant in their arguments. THIS IS ALWAYS WRONG (UNSOUND). Even when real languages implement it (e.g., Eiffel).
Covariant Example (C++)

But what happens if we make B::foo return a more general type?

class X {};
class Y : public X {};
class Z : public Y {};

class A {
public:
  virtual Y *foo() { return new Y(); }
};
class B : public A {
public:
  virtual X *foo() { return new X(); }  // 12
};

We can call B::foo with an X, Y, or Z, since B::foo takes an argument of type X (and \( Y \leq X \), \( Z \leq X \)).

Contravariant Example (C++)

But what if we modify B::foo to take an argument of type Z?

class X {};
class Y : public X {};
class Z : public Y {};
class A {
public:
  virtual Y *foo() { return new Y(); }
};
class B : public A {
public:
  virtual void foo(Y &y) { }  // 21
};

class B : public A {
public:
  virtual void foo(Z &z) { }  // 22
};

Notes:
- As always, elegantly handles arbitrarily large syntax (types)
- For other types, e.g., sums or pairs, would have more rules, deciding carefully about co/contravariance of each position

Contravariant Example (C++)

But what if we modify B::foo to take an argument of type Z?

class X {};
class Y : public X {};
class Z : public Y {};
class A {
public:
  virtual Y *foo() { return new Y(); }
};
class B : public A {
public:
  virtual void foo(Y &y) { }  // 21
};

class B : public A {
public:
  virtual void foo(Z &z) { }  // 22
};

t.cpp:7:14: note: overridden virtual function is here

t.cpp:12:23: note: passing argument to parameter 'z' here
virtual void foo(Z &z) { }
^
1 error generated.

Summary of subtyping rules

\[
\begin{align*}
\tau_1 &\leq \tau_2 & \tau_2 &\leq \tau_3 \\
\tau_1 &\leq \tau_3 & \tau &\leq \tau
\end{align*}
\]

\[
\{l_1;\tau_1;\ldots,l_n;\tau_n,l;\tau\} \leq \{l_1;\tau_1;\ldots,l_n;\tau_n\}
\]

\[
\{l_1;\tau_1;\ldots,l_{i-1};\tau_{i-1},l_i;\tau_i,\ldots,l_n;\tau_n\} \leq \{l_1;\tau_1;\ldots,l_l;\tau_l,l_{i-1};\tau_{i-1};\ldots,l_n;\tau_n\}
\]

\[
\{l_1;\tau_1;\ldots,l_i;\tau_i,\ldots,l_n;\tau_n\} \leq \{l_1;\tau_1;\ldots,l_l;\tau_l,\tau_i,\ldots,l_n;\tau_n\}
\]

\[
\{l_1;\tau_1;\ldots,l_l;\tau_l,\tau_i,\ldots,l_n;\tau_n\} \leq \{l_1;\tau_1;\ldots,l_l;\tau_l,\tau_i,\ldots,l_n;\tau_n\}
\]

\[
\tau_3 \leq \tau_1 & \tau_2 \leq \tau_4 & \tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4
\]

Notes:
- As always, elegantly handles arbitrarily large syntax (types)
- For other types, e.g., sums or pairs, would have more rules, deciding carefully about co/contravariance of each position
Maintaining soundness

Our Preservation and Progress Lemmas still "work" in the presence of subsumption

▶ So in theory, any subtyping mistakes would be caught when trying to prove soundness!

In fact, it seems too easy: induction on typing derivations makes the subsumption case easy:

▶ Progress: One new case if typing derivation $\cdot \vdash e : \tau$ ends with subsumption. Then $\cdot \vdash e : \tau'$ via a shorter derivation, so by induction a value or takes a step.

▶ Preservation: One new case if typing derivation $\cdot \vdash e : \tau$ ends with subsumption. Then $\cdot \vdash e : \tau'$ via a shorter derivation, so by induction if $e \rightarrow e'$ then $\cdot \vdash e' : \tau'$. So use subsumption to derive $\cdot \vdash e' : \tau$.

Hmm...

Boyana Norris

Ah, Canonical Forms

That’s because Canonical Forms is where the action is:

▶ If $\cdot \vdash v : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$, then $v$ is a record with fields $l_1, \ldots, l_n$

▶ If $\cdot \vdash v : \tau_1 \rightarrow \tau_2$, then $v$ is a function

We need these for the "interesting" cases of Progress

Now have to use induction on the typing derivation (may end with many subsumptions) and induction on the subtyping derivation (e.g., "going up the derivation" only adds fields)

▶ Canonical Forms is typically trivial without subtyping; now it requires some work

Note: Without subtyping, Preservation is a little "cleaner" via induction on $e \rightarrow e'$, but with subtyping it’s much cleaner via induction on the typing derivation

▶ That’s why we did it that way

Boyana Norris

A matter of opinion?

If subsumption makes well-typed terms get stuck, it is wrong

We might allow less subsumption (e.g., for efficiency), but we shall not allow more than is sound

But we have been discussing "subset semantics" in which $e : \tau$ and $\tau \leq \tau'$ means $e$ is a $\tau'$

▶ There are "fewer" values of type $\tau$ than of type $\tau'$, but not really

Very tempting to go beyond this, but you must be very careful...

But first we need to emphasize a really nice property of our current setup: Types never affect run-time behavior

Boyana Norris

Erasure

A program type-checks or does not. If it does, it evaluates just like in the untyped $\lambda$-calculus. More formally, we have:

1. Our language with types (e.g., $\lambda x : \tau. e$, $A_{\tau_1 + \tau_2}(e)$, etc.) and a semantics

2. Our language without types (e.g., $\lambda x. e$, $A(e)$, etc.) and a different (but very similar) semantics

3. An erasure metafunction from first language to second

4. An equivalence theorem: Erasure commutes with evaluation

This useful (for reasoning and efficiency) fact will be less obvious (but true) with parametric polymorphism

Boyana Norris

Coercion Semantics

Wouldn’t it be great if...

▶ int $\leq$ float

▶ int $\leq \{l_1 : \text{int}\}$

▶ $\tau \leq \text{string}$

▶ we could "overload the cast operator"

For these proposed $\tau \leq \tau'$ relationships, we need a run-time action to turn a $\tau$ into a $\tau'$

▶ Called a coercion

Could use $\text{float_of_int}$ and similar but programmers whine about it

Boyana Norris

Implementing Coercions

If coercion $C$ (e.g., $\text{float_of_int}$) "witnesses" $\tau \leq \tau'$ (e.g., int $\leq$ float), then we insert $C$ where $\tau$ is subsumed to $\tau'$

So translation to the untyped language depends on where subsumption is used. So it’s from typing derivations to programs.

But typing derivations aren’t unique: uh-oh

Example 1:

▶ Suppose int $\leq$ float and $\tau \leq \text{string}$

▶ Consider $\cdot \vdash \text{print_string}(34) : \text{unit}$

Example 2:

▶ Suppose int $\leq \{l_1 : \text{int}\}$

▶ Consider $34 == 34$, where $==$ is equality on ints or pointers

Boyana Norris
Coherence

Coercions need to be coherent, meaning they don’t have these problems.

More formally, programs are deterministic even though type checking is not—any typing derivation for $e$ translates to an equivalent program.

Alternately, can make (complicated) rules about where subsumption occurs and which subtyping rules take precedence.

- Hard to understand, remember, implement correctly
- It’s a mess...

C++

Semi-Example: Multiple inheritance a la C++

```c++
class C2 {};
class C3 {};
class C1 : public C2, public C3 {};
class D {
public:
    int f(class C2) { return 0; }
    int f(class C3) { return 1; }
};
int main() { return D().f(C1()); }
```

Note: A compile-time error “ambiguous call”

Note: Same in Java with interfaces (“reference is ambiguous”)

Upcasts and Downcasts

- “Subset” subtyping allows “upcasts”
- “Coercive subtyping” allows casts with run-time effect
- What about “downcasts”?

That is, should we have something like:

```c++
if_hastype(\tau, e_1) \text{ then } x \cdot e_2 \text{ else } e_3
```

Roughly, if at run-time $e_1$ has type $\tau$ (or a subtype), then bind it to $x$ and evaluate $e_2$. Else evaluate $e_3$. Avoids having exceptions.

- Not hard to formalize

Downcasts

Can’t deny downcasts exist, but here are some bad things about them:

- Types don’t erase – you need to represent $\tau$ and $e_1$’s type at run-time. (Hidden data fields)
- Breaks abstractions: Before, passing $\{l_1 = 3, l_2 = 4\}$ to a function taking $\{l_1 : \text{int}\}$ hid the $l_2$ field, so you know it doesn’t change or affect the callee.

Some better alternatives:

- Use ML-style datatypes — the programmer decides which data should have tags
- Use parametric polymorphism — the right way to do container types (not downcasting results)

Parametric Polymorphism

Done with subtyping.

Now: Parametric polymorphism

When type inference determines than an expression is valid for any type it is automatically made polymorphic. In OCaml, the polymorphic types in type expressions are denoted ‘a, ’b, ’c and so on. For example, the following function reverses the order of the elements in a 2-tuple (pair) and can be applied to pairs of values of any type:

```ocaml
# let rev2 (x, y) = (y, x);;
val rev2 : 'a * 'b -> 'b * 'a = <fun>
# rev2 (1, 2);;
- : int * int = (2, 1)
# rev2 ("hello", 1);
- : string * int = ("hello", 1)
```

Goal

Understand what this interface means and why it matters:

```ocaml
type 'a mylist;
val mt_list : 'a mylist
val cons : 'a -> 'a mylist -> 'a mylist
val decons : 'a mylist -> (('a * 'a mylist) option)
val length : 'a mylist -> int
val map : ('a -> 'b) -> 'a mylist -> 'b mylist
```

From two perspectives:

1. Library: Implement code to this partial specification
2. Client: Use code written to this partial specification
What The Client Likes

1. Library is reusable. Can make:
   ◦ Different lists with elements of different types
   ◦ New reusable functions outside of library, e.g.:
     ```
     val twocons : 'α -> 'a -> 'a mylist -> 'a mylist
     ```

2. Easier, faster, more reliable than subtyping
   ◦ No downcast to write, run, maybe-fail (cf. Java 1.4 Vector)

3. Library must “behave the same” for all "type instantiations"!
   ◦ 'a and 'b held abstract from library
   ◦ E.g., with built-in lists: If foo has type 'a list → int, then
     ```
     foo [1;2;3] and foo [(6,4);(7,2);(9,2)] are totally equivalent!
     ```
   (Never true with downcasts)
   ◦ In theory, means less (re-)integration testing
   ◦ Proof is beyond this course, but not much

Start simpler

The interface has a lot going on:

1. Element types held abstract from library
2. List type (constructor) held abstract from client
3. Reuse of type variables "makes connections" among expressions of abstract types
4. Lists need some form of recursive type

This lecture considers just (1) and (3)
   ▶ First using a formal language with explicit type abstraction
   ▶ Then mention differences from ML

Note: Much more interesting than "not getting stuck"

What the Library Likes

1. Reusability — For same reasons client likes it
2. Abstraction of mylist from clients
   ◦ Clients must “behave the same” for all equivalent implementations, even if "hidden definition" of ‘a mylist changes
   ◦ Clients typechecked knowing only there exists a type constructor mylist
   ◦ Unlike Java, C++, R5RS Scheme, no way to downcast a t mylist to, e.g., a pair

Informal semantics

1. Δα. e: A value that, when used, runs e (with some τ for α)
   ◦ To type-check e, know α is one type, but not which type

2. e[τ]: Evaluate e to some Δα. e’ and then run e’
   ◦ With τ for α, but the choice of τ is irrelevant at run-time
   ◦ τ used for type-checking and proof of Preservation

3. Types can use type variables α, β, etc., but only if they’re in scope (just like term variables)
   ◦ Type-checking will be Δ; Γ ⊢ e : τ using Δ to know what type variables are in scope in e
   ◦ In universal type ∀α.τ, can also use α in τ

Operational semantics

Small-step, CBV, left-to-right operational semantics:
   ◦ Note: Δα. e is a value

```
\begin{align*}
  e \rightarrow e' \\
  e_1 e_2 \rightarrow e'_1 e_2 \\
  e_2 \rightarrow e'_2 \\
  (\lambda x:τ. e) v \rightarrow e[v/x]
\end{align*}
```

Old:

New:

```
\begin{align*}
  e[τ] \rightarrow e'[τ] \\
(Δα. e)[τ] \rightarrow e[τ/α]
\end{align*}
```

Plus now have 3 different kinds of substitution, all defined in straightforward capture-avoiding way:

- e1[e2/x] (old)
- e[τ'/α] (new)
- τ[τ'/α] (new)
The Whole Language, Called System F

```
e ::= c | x | λx:τ. e | e e | Λα.e | e[τ]
τ ::= int | τ → τ | α | ∀α.τ
v ::= c | λx:τ. e | Λα.e
Γ ::= · | Γ, x:τ
Δ ::= ·
```

```
e → e'
e e2 → e' e2
v e → v e'
e[τ] → e'[τ]
(λx:τ. e) v → e[v/x]
(Λα.e)[τ] → e[τ/α]
```

```
Δ; Γ ⊢ x : Γ(x) → Δ; Γ ⊢ e : int
Δ; Γ ⊢ e : τ1
→ Δ; Γ ⊢ ∀α.τ1 → Δ; Γ ⊢ e[τ2] : τ1[τ2/α]
```

Examples

```
Example (using addition):
(Λα. Λβ. λx:α. λf:α → β. f x) [int] [int] 3 (λy : int. y + y)
→ (Λβ. λx : int. λf:int → β. f x) [int] 3 (λy : int. y + y)
→ (λx : int. λf:int → int. f x) 3 (λy : int. y + y)
→ (λy : int. y + y) 3
→ 3 + 3
→ 6
```

Type System, part 1

 Mostly just need to be picky about "no free type variables"

- Typing judgment has the form \( Δ; \Gamma \vdash e : τ \)  
  (whole program \( \cdot \vdash e : τ \))
  - Next slide
- Uses helper judgment \( Δ ⊢ τ \)
  - "all free type variables in \( τ \) are in \( Δ \)"

```
Δ ⊢ τ
\( \alpha ∈ Δ \)
\( Δ ⊢ τ \)
\( Δ ⊢ τ_1 \)
\( Δ ⊢ τ_2 \)
\( Δ, α ⊢ τ \)
```

Rules are boring, but trust me, allowing free type variables is a pernicious source of language/compiler bugs

Example

```
Example (using addition):
(Λα. Λβ. λx:α. λf:α → β. f x) [int] [int] 3 (λy : int. y + y)
```

(The typing derivation is rather tall and painful, but just a syntax-directed derivation by instantiating the typing rules)

Type System, part 2

Old (with one technical change to prevent free type variables):

```
\( Δ; Γ \vdash x : Γ(x) \)
\( Δ; Γ \vdash c : \text{int} \)
\( Δ; Γ \vdash e : τ_1 \)
\( Δ; Γ \vdash e : τ_2 \)
\( Δ ⊢ τ_1 \)
\( Δ ⊢ τ_2 \)

New:

```
\( Δ, α; Γ \vdash e : τ_1 \)
\( Δ, α; Γ \vdash e : ∀α.τ_1 \)
\( Δ ⊢ τ_2 \)
```

The Whole Language, Called System F
More Examples

Let \( \text{apply1} = \Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda f : \alpha \rightarrow \beta. f \ x \)
- \( \text{apply1} \) has type \( \forall \alpha. \forall \beta. \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta \)
- \( \cdot; g : \text{int} \rightarrow \text{int} \vdash (\text{apply1} \ ... (f \ x). \)
- \( \text{twice} \) has type \( \forall \alpha. \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha \)
- Cannot be made more polymorphic

Example: A process \( e \) should not access files it did not open (\( \text{fopen} \) can check permissions)
Let \( e_2 = \text{erase}(e_1) \)
"Erasure and evaluation commute"

Moral of Example
In simply-typed lambda-calculus, type safety just means not getting stuck
With type abstraction, it enables secure interfaces!
Suppose we (the system library) implement file-handles as ints.
Then we instantiate \( \alpha \) with \( \text{int} \), but untrusted code cannot tell
Memory safety is a necessary but insufficient condition for language-based enforcement of strong abstractions

Metatheory
- Safety: Language is type-safe
  - Need a Type Substitution Lemma
- Termination: All programs terminate
  - Surprising — we saw id \([ \tau ] \) id
- Parametricity, a.k.a. theorems for free
  - Example: If \( \cdot; \vdash e : \forall \alpha. \forall \beta. (\alpha \times \beta) \rightarrow (\beta \times \alpha) \), then \( e \) is equivalent to \( \Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda y : \beta. (x,2,y,1) \).
  - Every term with this type is the swap function!!
    - Intuition: \( e \) has no way to make an \( \alpha \) or a \( \beta \) and it cannot tell what \( \alpha \) or \( \beta \) are or raise an exception or diverge...
- Erasure: Types do not affect run-time behavior
  - Note: Mutation "breaks everything"
    - depth subtyping: hw4, termination: hw3, parametricity: hw5

Security from safety?
Example: A process \( e \) should not access files it did not open (\( \text{fopen} \) can check permissions)
Require an untrusted process \( e \) to type-check as follows:
\( \cdot; \vdash e : \forall \alpha. \{ \text{fopen} : \text{string} \rightarrow \alpha, \text{fread} : \alpha \rightarrow \text{int} \} \rightarrow \text{unit} \)
This type ensures that a process won’t "forge a file handle" and pass it to \( \text{fread} \)
So \( \text{fread} \) doesn’t need to check (faster), file handles don’t need to be encrypted (safer), etc.

Are types used at run-time?
We said polymorphism was about "many types for same term", but for clarity and easy checking, we changed:
- The syntax via \( \Delta \)
- The operational semantics via type substitution
- The type system via \( \Delta \)

Claim: The operational semantics did not "really" change; types need not exist at run-time
More formally: Erasing all types from System F produces an equivalent program in the untyped lambda calculus
Strengthened induction hypothesis: If \( e \rightarrow e_1 \) in System F and \( \text{erase}(e) \rightarrow e_2 \) in untyped lambda-calculus, then \( e_2 = \text{erase}(e_1) \)
"Erasure and evaluation commute"
Erasure

Erasure is easy to define:

\[
\begin{align*}
\text{erase}(c) &= c \\
\text{erase}(x) &= x \\
\text{erase}(e_1 e_2) &= \text{erase}(e_1) \ \text{erase}(e_2) \\
\text{erase}(\lambda x:\tau. e) &= \lambda x. \ \text{erase}(e) \\
\text{erase}(\Lambda \alpha. e) &= \lambda. \ \text{erase}(e) \\
\text{erase}(e[\tau]) &= \text{erase}(e) 0
\end{align*}
\]

In pure System F, preserving evaluation order isn’t crucial, but it is with fix, exceptions, mutation, etc.

Connection to reality

System F has been one of the most important theoretical PL models since the 1970s and inspires languages like ML.

But you have seen ML polymorphism and it looks different. In fact, it is an implicitly typed restriction of System F.

These two qualifications ((1) implicit, (2) restriction) are deeply related.

Restrictions

- All types have the form \(\forall \alpha_1, \ldots, \alpha_n. \tau\) where \(n \geq 0\) and \(\tau\) has no \(\forall\). (Prenex-quantification; no first-class polymorphism.)
- Only let (rec) variables (e.g., \(x\) in \(\text{let } x = e_1 \ \text{in } e_2\)) can have polymorphic types. So \(n = 0\) for function arguments, pattern variables, etc. (Let-bound polymorphism)
  - So cannot (always) desugar let to \(\lambda\) in ML
- In \(\text{let rec } f \ x = e_1 \ \text{in } e_2\), the variable \(f\) can have type \(\forall \alpha_1, \ldots, \alpha_n. \tau_1 \rightarrow \tau_2\) only if every use of \(f\) in \(e_1\) instantiates each \(\alpha_i\) with \(\alpha_i\). (No polymorphic recursion)
- Let variables can be polymorphic only if \(e_1\) is a "syntactic value"
  - A variable, constant, function definition, ...
  - Called the "value restriction" (relaxed partially in OCaml)

Why?

ML-style polymorphism can seem weird after you have seen System F. And the restrictions do come up in practice, though tolerable.

- Type inference for System F (given untyped \(e\), is there a System F term \(e'\) such that \(\text{erase}(e') = e\) ) is undecidable (1995)
- Type inference for ML with polymorphic recursion is undecidable (1992)
- Type inference for ML is decidable and efficient in practice, though pathological programs of size \(O(n)\) and run-time \(O(n)\) can have types of size \(O(2^n)\)
- The type inference algorithm is unsound in the presence of ML-style mutation, but value-restriction restores soundness
  - Based on unification

Recovering lost ground?

Extensions to the ML type system to be closer to System F:

- Usually require some type annotations
- Are judged by:
  - Soundness: Do programs still not get stuck?
  - Conservatism: Do all (or most) old ML programs still type-check?
  - Power: Does it accept many more useful programs?
  - Convenience: Are many new types still inferred?