CIS 624: Structure of Programming Languages
Lecture 15 — Subtyping, Parametric Polymorphism, System F
Boyana Norris
2015

An overloaded PL word
Polymorphism means many things . . .

- Ad hoc polymorphism: $e_1 + e_2$ in SML $<$ C $<$ Java $<$ C++
- Ad hoc, cont’d: Maybe $e_1$ and $e_2$ can have different run-time types and we choose the $+$ based on them
- Parametric polymorphism: e.g., $\Gamma \vdash \lambda x. x : \forall \alpha : \alpha \rightarrow \alpha$ or with explicit types: $\Gamma \vdash \Lambda \alpha. \lambda x. x : \forall \alpha : \alpha \rightarrow \alpha$
  (which "compiles" i.e. "erases" to $\lambda x. x$)
- Subtype polymorphism: new Vector().add(new C()) is legal Java because new C() has types Object and C
  ...and nothing.
  (More precise terms: "static overloading," "dynamic dispatch," "type abstraction," and "subtyping")

Today
This first part of this lecture is about subtyping

- Let more terms type-check without adding any new operational behavior
  - But at end consider coercions
- Continue using STLC as our core model
- Complementary to type variables which we will do later
  - Parametric polymorphism ($\forall$), a.k.a. generics
  - First-class ADTs ($\exists$)

Being Less Restrictive

"Will a $\lambda$ term get stuck?" is undecidable, so a sound, decidable type system can always be made less restrictive

An "uninteresting" rule that is sound but not "admissable":

$$\Gamma \vdash e_1 : \tau$$
$$\Gamma \vdash \text{if } \text{true } e_1 \text{ e2 } : \tau$$

We’ll study ways to give one term many types ("polymorphism")

Fact: The version of STLC with explicit argument types ($\lambda x : \tau. e$) has no polymorphism:
If $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$, then $\tau_1 = \tau_2$

Fact: Even without explicit types, many "reuse patterns" do not tran type-check. Example: ($\lambda f. (f 0, f \text{true}))((\lambda x. (x, x))$
(evaluates to ((0, 0), (true, true)))

Records
We’ll use records to motivate subtyping:

$$e ::= \ldots | \{l_1 = e_1, \ldots, l_n = e_n\} | e.l$$
$$\tau ::= \ldots | \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$$

$\Gamma \vdash e_1 : \tau_1 \ldots \Gamma \vdash e_n : \tau_n$ labels distinct

$\Gamma \vdash \{l_1 = e_1, \ldots, l_n = e_n\} : \{l_1 : \tau_1, \ldots, l_n : \tau_n\}$

$\Gamma \vdash e : \{l_1 : \tau_1, \ldots, l_n : \tau_n\} \quad 1 \leq i \leq n$

Suggests subtyping:

$$\tau_1 \leq \tau_2$$

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

And one one new type-checking rule: Subsumption

Subsumption

$$\frac{}{\Gamma \vdash e : \tau' \quad \tau' \leq \tau}$$
Now it type-checks

\[ \forall x : \{(1\text{int}, 2\text{int})\} : x.l_1 + x.l_2 : \text{int} \]

\[ \forall (\lambda x: \{l_1: \text{int}, l_2: \text{int}\}. x.l_1 + x.l_2) : \{l_1: \text{int}, l_2: \text{int}\} \rightarrow \text{int} \]

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

Transitivity

Subtyping is always transitive, so add a rule for that:

\[ \tau_1 \leq \tau_2, \tau_2 \leq \tau_3 \rightarrow \tau_1 \leq \tau_3 \]

Or just use the subsumption rule multiple times. Or both.

In any case, type-checking is no longer syntax-directed: There may be 0, 1, or many different derivations of \( \Gamma \vdash e : \tau \)
- And also potentially many ways to show \( \tau_1 \leq \tau_2 \)

Hopefully we could define an algorithm and prove it “answers yes” if and only if there exists a derivation

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: Efficiency

With our semantics, width and permutation subtyping make perfect sense

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

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...

\[ f_1 : \{l_1 : \text{int}\} \rightarrow \text{int} \quad f_2 : \{l_2 : \text{int}\} \rightarrow \text{int} \]

\[ x_1 = \{l_1 = 0, l_2 = 0\} \quad x_2 = \{l_2 = 0, l_3 = 0\} \]

\[ f_1(x_1) \quad f_2(x_1) \quad f_2(x_2) \]

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

\textit{Named types} can avoid this, but make code less flexible
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 \( Y \).
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 ≤ X, Z ≤ 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(Z &z) { }
};

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

Summary of subtyping rules

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

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

1 error generated.

Covariant Example (C++)

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

g++ -W -Wall -ansi -pedantic -c t.cpp
t.cpp:12:14: error: return type of virtual function 'foo' is not covariant with the return type of the function it overrides ('X *' is not derived from 'Y *')
virtual X *foo() { return new X(); } // 12
^
1 error generated.
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...

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 $e : \tau$ 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*

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

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

Coercion Semantics

Wouldn’t it be great if...

▶ int $\leq$ float

▶ int $\leq \{l_1 : \text{int}\}$

▶ $\tau$ $\leq$ 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 float_of_int and similar but programmers whine about it

Implementing Coercions

If coercion $C$ (e.g., 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 $\tau \leq$ 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
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++

```
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:

```
if_hastype(\( \tau, e_1 \)) then \( x, e_2 \) 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:

```
# 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:

```
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 : 'a -> '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

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
   - 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. $\Lambda\alpha.e$: A value that, when used, runs $e$ (with some $\tau$ for $\alpha$)
   - To type-check $e$, know $\alpha$ is one type, but not which type
   - $\tau$ used for type-checking and proof of Preservation

2. $e[\tau]$:
   - Evaluate $e$ to some $\Lambda\alpha.e$. $e'$ and then run $e'$
   - With $\tau$ for $\alpha$, but the choice of $\tau$ is irrelevant at run-time
   - $\tau$ used for type-checking and proof of Preservation

3. Types can use type variables $\alpha$, $\beta$, etc., but only if they're in
   scope (just like term variables)
   - Type-checking will be $\Delta; \Gamma \vdash e : \tau$ using $\Delta$ to know what
     type variables are in scope in $e$
   - In universal type $\forall\alpha.\tau$, can also use $\alpha$ in $\tau$

Syntax

$$
e ::= c | x | \lambda x:\tau. e | e e | \Lambda\alpha.e | e[\tau]$
$$
$$
\tau ::= \text{int} | \tau \rightarrow \tau | \alpha | \forall \alpha.\tau$
$$
$$
v ::= c | \lambda x:\tau. e | \Lambda\alpha.e$
$$
$$
\Gamma ::= \cdot | \Gamma, x:\tau$
$$
$$
\Delta ::= \cdot | \Delta, \alpha$
$$

New things:
- Terms: Type abstraction and type application
- Types: Type variables and universal types
- Type contexts: what type variables are in scope

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"

Operational semantics

Small-step, CBV, left-to-right operational semantics:
- Note: $\Lambda\alpha.e$ is a value

Old:
$$
ed_1 \rightarrow ed_1' \quad ed_2 \rightarrow ed_2' \quad v e_2 \rightarrow v e_2' \quad (\lambda x:\tau. e) v \rightarrow e[v/x]$
$$

New:
$$
ed \rightarrow ed' \quad \Lambda\alpha.e[\tau] \rightarrow e[\tau/\alpha]$
$$

Plus now have 3 different kinds of substitution, all defined in
straightforward capture-avoiding way:
- $e_1[e_2/x]$ (old)
- $e[\tau'/\alpha]$ (new)
- $\tau[\tau'/\alpha]$ (new)
Example

Example (using addition):

\[ \Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda f : \beta. \ f \ x \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda y : \text{int}. \ y + y) \]
\[ \rightarrow (\Lambda \alpha. \lambda x : \text{int}. \lambda f : \text{int}. \ f \ x) \ [\text{int}] \ [\text{int}] \ 3 \ (\lambda y : \text{int}. \ y + y) \]
\[ \rightarrow \ (\lambda f : \text{int}. \ f \ 3) \ (\lambda y : \text{int}. \ y + y) \]
\[ \rightarrow \ (\lambda y : \text{int}. \ y + y) \ 3 \]
\[ \rightarrow \ 3 + 3 \]
\[ \rightarrow \ 6 \]

Type System, part 2

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

\[ \Delta; \Gamma \vdash x : \Gamma(x) \quad \Delta; \Gamma \vdash c : \text{int} \]
\[ \Delta; \Gamma; e_1 : \tau_1 ; e_2 : \tau_2 \quad \Delta \vdash \tau_1 \rightarrow \tau_2 \]
\[ \Delta; \Gamma; \alpha : \tau_1 \quad \Delta; \Gamma; e_2 : \tau_2 \quad \Delta; \Gamma; e_2 : (\forall \alpha. \tau_1) \rightarrow \tau_2 \]

New:

\[ \Delta; \Gamma; \alpha : \tau_1 \quad \Delta; \Gamma; e_2 : (\forall \alpha. \tau_1) \rightarrow \tau_2 \]
\[ \Delta; \Gamma; e_2 : (\forall \alpha. \tau_1) [\tau_2/\alpha] \]

The Whole Language, Called System F

\[ e ::= c | x | \lambda x : \tau. \ e | e \ e | \Lambda \alpha. \ e | e[\tau] \]
\[ \tau ::= \text{int} | \tau \rightarrow \tau | \alpha | \forall \alpha. \tau \]
\[ v ::= c | \lambda x : \tau. \ e | \Lambda \alpha. \ e \]
\[ \Gamma ::= \cdot | \Gamma, \alpha : \tau \]
\[ \Delta ::= \cdot | \Delta, \alpha \]
\[ e \rightarrow e' \]
\[ e_2 \rightarrow e'_2 \]
\[ v \rightarrow v' \]
\[ e[\tau] \rightarrow e'[\tau] \]
\[ (\lambda x : \tau. \ e) v \rightarrow e[v/x] \]
\[ (\Lambda \alpha. \ e)[\tau] \rightarrow e[\tau/\alpha] \]
\[ \Delta; \Gamma \vdash x : \Gamma(x) \]
\[ \Delta; \Gamma \vdash c : \text{int} \]
\[ \Delta; \Gamma; e_1 : \tau_1 ; e_2 : \tau_2 \quad \Delta \vdash \tau_1 \rightarrow \tau_2 \]
\[ \Delta; \Gamma; e_1 : \tau_1 \rightarrow \tau_2 \]
\[ \Delta; \Gamma; e_2 : \tau_2 \]
\[ \Delta; \Gamma; e_2 : (\forall \alpha. \tau_1) \rightarrow \tau_2 \]

Examples

An overly simple polymorphic function...

Let id = \Lambda \alpha. \lambda x : \alpha. \ x

\[ \text{id} \]
\[ \text{id} \ has \ type \ (\forall \alpha. \alpha) \rightarrow \alpha \]
\[ \text{id} [\text{int}] \ has \ type \ [\text{int}] \rightarrow [\text{int}] \]
\[ \text{id} [\text{int} \rightarrow \text{int}] \ has \ type \ ([\text{int} \rightarrow \text{int}] \rightarrow \text{int} \rightarrow \text{int}) \]
More Examples

Let \( \text{apply1} = \Lambda \alpha. \Lambda \beta. \lambda x : \alpha. \lambda f : \alpha \to \beta. f \ x \)

- \( \text{apply1} \) has type \( \forall \alpha. \forall \beta. \alpha \to (\alpha \to \beta) \to \beta \)
- \( \vdash g : \text{int} \to \text{int} \vdash \text{apply1} \text{ [int]} \text{ [int]} \text{ [3]} \to \text{int} \)

Let \( \text{apply2} = \Lambda \alpha. \lambda x : \alpha. \lambda f : \alpha \to \beta. f \ x \)

- \( \text{apply2} \) has type \( \forall \alpha. \forall \beta. (\alpha \to \beta) \to \beta \)
  (impossible in ML)
- \( \vdash g : \text{int} \to \text{string}, h : \text{int} \to \text{int} \vdash \text{let } z = \text{apply2 [int]} \text{ in } z \ 3 \text{ [int]} h \text{ [string]} g : \text{string} \)

Let \( \text{twice} = \Lambda \alpha. \lambda x : \alpha. \lambda f : \alpha \to \alpha. f \ (f \ x) \).

- \( \text{twice} \) has type \( \forall \alpha. \alpha \to (\alpha \to \alpha) \to \alpha \)
- Cannot be made more polymorphic

What next?

Having defined System F...

- Metatheory (what properties does it have)
- What (else) is it good for
- How/why ML is more restrictive and implicit

Security from safety?

Example: A process \( e \) should not access files it did not open
(fopen can check permissions)

Require an untrusted process \( e \) to type-check as follows:

\[ \vdash e : \forall \alpha. \{ \text{fopen} : \text{string} \to \alpha, \text{fread} : \alpha \to \text{int} \} \to \text{unit} \]

This type ensures that a process won’t “forge a file handle” and pass it to fread

So 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 \( \Lambda \alpha. \ e \) and \( e \ [\tau] \)
- 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 \( e \to e_1 \) in System F and \( \text{erase} (e) \to e_2 \) in untyped lambda-calculus, then \( e_2 = \text{erase} (e_1) \)

“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 \([r] \) id
- Parametricity, a.k.a. theorems for free
  - Example: If \( \vdash e : \forall \alpha. \forall \beta. (\alpha \star \beta) \to (\beta \star \alpha) \), then \( e \) is equivalent to \( \Lambda \alpha. \Lambda \beta. \lambda x : \alpha \star \beta. (x \cdot 2, x \cdot 1) \)
    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

Erasure and evaluation commute

“Erasure and evaluation commute”
### Erasure

Erasure is easy to define:

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

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 \( O(2^{2^n}) \)

- 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?