

1. Recall Let-Polymorphism

In simply-typed lambda-calculus, we can leave out ALL type annotations:

→ insert new type variables
 → do type reconstruction (using unification)

In this way, changing the let-rule, we obtain

- Let-Polymorphism
- \rightarrow Simple form of polymorphism
- \rightarrow Introduced by [Milner 1978] in ML
- ightarrow also known as Damas-Milner polymorphism
- → in ML, basis of powerful generic libraries (e.g., lists, arrays, trees, hash tables, ...)

1. Recall Let-Polymorphism $\begin{array}{c} \Gamma \vdash t_1:T_1 & \Gamma \vdash [x \Rightarrow t_1]t_2:T_2 \\ \hline \Gamma \vdash let x = t_1 \text{ in } t_2:T_2 \end{array} \\ \hline \end{array}$ $\begin{array}{c} \left[\begin{array}{c} \text{let double } = \lambda x. \lambda y, \ x(x(y)) \ \text{in} \\ \\ \\ \text{let double } = \lambda u \cdot \lambda y, \ x(x(y)) \ \text{in} \\ \\ \\ \\ \\ \\ \\ \end{array} \right] \\ \hline \end{array}$ $\begin{array}{c} \text{let double } (\lambda x: \text{int. } x + 2) \ 2 \ \text{in} \\ \\ \\ \\ \\ \\ \\ \\ \\ \\ \end{array} \\ \hline \end{array}$ $\begin{array}{c} \text{CAN be typed now!! Because the new let rule creates two copies of double, and the rule for abstraction assigns a$ *different* $type variable to each one. \end{array}$

1. Recall Let-Polymorphism

Limits of Let-Polymorphism?

- → Only let-bound variables can be used polymorphically!
 → NOT lambda-bound variables
- Ex.: let $f = \lambda g$ g(1) ... g(true) ... in { $f(\lambda x.x)$ }
- is not typable: when typechecking the def. of f, g has type X (fresh) Which is then constrained by $X = int \rightarrow Y$ and $X = bool \rightarrow Z$.

Functions cannot take polymorphic functions as parameters.

(= no polymorphic arguments!)

2. System F

Aka polymorphic lambda-calculus or second-order lambda-calculus.

→ do lambda-abstraction over type variables, define functions over types

Invented by

- → Girard (1972) motivated by logics
- → Reynolds (1974) motivated by programming.

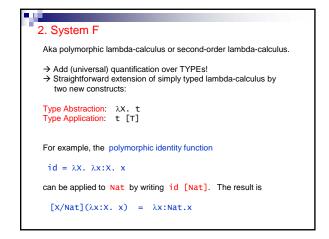
2. System F

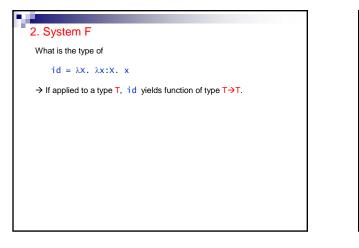
Aka polymorphic lambda-calculus or second-order lambda-calculus.

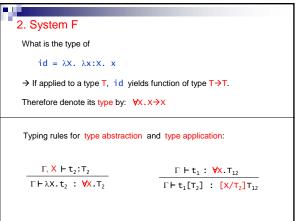
 → Add (universal) quantification over TYPEs!
 → Straightforward extension of simply typed lambda-calculus by two new constructs:

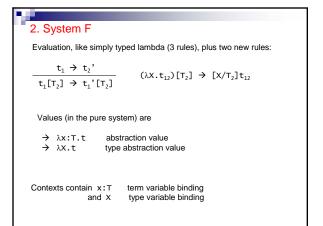
For example, the polymorphic identity function

id = λX . λx :X. x









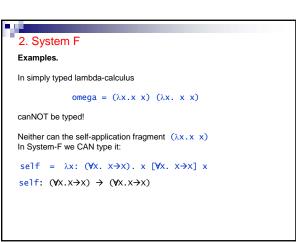
	System F mples.
Poly	morphic identity function: $id = \lambda X \cdot \lambda x : X \cdot x$
Appl	y it:
= (→ [→ (d [Nat] 5 (\x.\x:x. x) [Nat] 5 [x/Nat](\x:x. x) 5 [\x:Nat. x) 5 [x/5](x)
As w	we saw, the type of id is $\forall X. X \rightarrow X.$
Can	you find a function different from id, with the SAME TYPE??

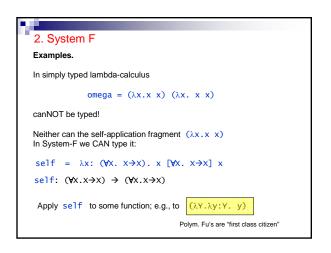
2. System F Examples. Polymorphic doubling function: double = λX. λf:x→X. λa:X. f (f a) double [Nat] (λx:Nat. succ(succ(x))) 3

quadruple = λX . double [X \rightarrow X] (double [X])

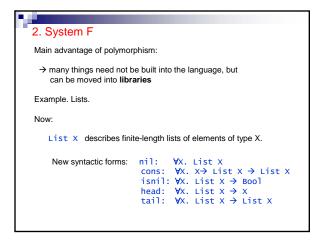
What's the type of quadruple?

 $\rightarrow 7$





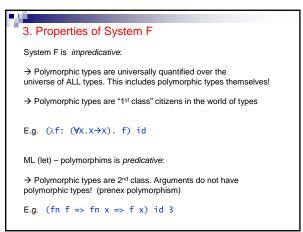
2. System F Main advantage of polymorphism	
Main advantage of polymorphism:	
→ many things need not be built can be moved into libraries	into the language, but
Example. Lists.	
Before:	
For a type T: List T describes	s finite-length lists of elements from T.
isn hea	[T] s[T] t1 t2 i1[T] t d[T] t 1[T] t

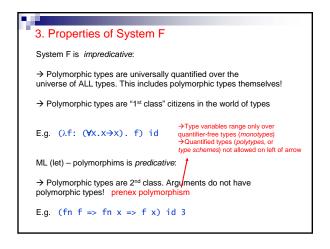


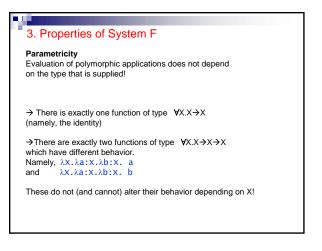
2. System F

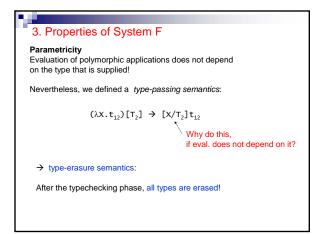
Now we can build a library of polymorphic operations on lists. For example, a polymorphic ${\tt map}$ function.

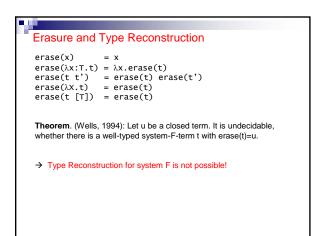
```
 \begin{array}{l} \mathsf{map} = \lambda X.\lambda Y. \\ \lambda f: X \rightarrow Y. \\ (fix (\lambda m: List X \rightarrow List Y. \\ \lambda 1: List X. \\ if isnil[X] 1 then nil[Y] \\ else \ cons[Y](f \ (head[X] 1)) \\ (m \ (tail[X] 1)))) \end{array}
```

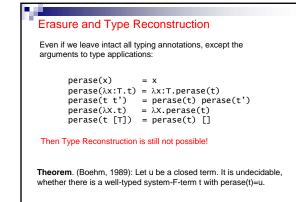


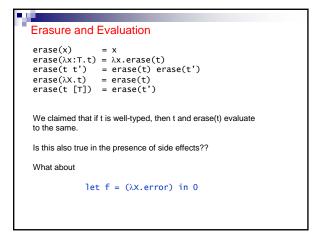


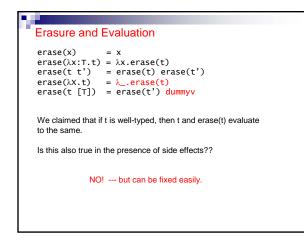


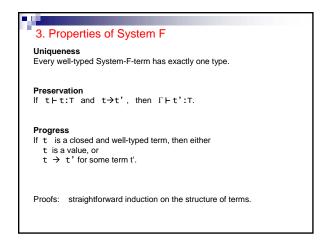












3. Properties of System F

Normalization

Every well-typed System-F-term t is normalizing, i.e.,

∃t': t →^{*} t' **≯**

Proof: very hard (Girard's PhD thesis, 1972) → later simplified to about 5 pages

Surprising: normalization holds even though MANY things can be coded in System F!

Can the (erased) term $(\lambda x.x x)(\lambda x.x x)$ be typed in System F?



Every well-typed System-F-term t is normalizing, i.e.,

∃t': t →^{*} t' ≯

Proof: very hard (Girard's PhD thesis, 1972) → later simplified to about 5 pages

Surprising: normalization holds even though MANY things can be coded in System F!

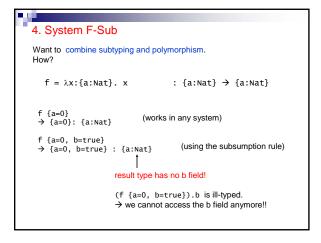
Can the (erased) term $(\lambda x.x x)(\lambda x.x x)$ be typed in System F?

→ This can even be proved directly! Do EXERCISE 23.6.3 in TAPL!

3. Properties of System F

When is (partial) type reconstruction possible??

- $\rightarrow~$ First-class existential types (e.g., using ML's datatype mechanism)
- $\rightarrow\,$ Add to that universal quantifiers which may appear in annotations of function arguments
- In the presence of subtyping:
- → Local type inference



4. System F-Sub

Use polymorphic identity fpoly instead of f:

$$\begin{split} &\mathsf{f} = \lambda x \colon \{a:\mathsf{Nat}\}. \ x \ : \ \{a:\mathsf{Nat}\} \rightarrow \{a:\mathsf{Nat}\} \\ &\mathsf{fpoly} = \lambda x. \ \lambda x \colon x \ : \ (\forall x. x \rightarrow x) \end{split}$$

```
4. System F-Sub
Use polymorphic identity fpoly instead of f:
f = \lambda x: \{a: Nat\}. x : \{a: Nat\} \rightarrow \{a: Nat\}
fpoly = \lambda x. \lambda x: x. x : (\forall x. x \rightarrow x)
fpoly [{a:Nat, b:Bool}] {a=0, b=true}
```

 \rightarrow {a=0, b=true} : {a:Nat, b:Bool}

HURRA!

4. System F-Sub

 $f2 = \lambda x: \{a:Nat\}. \{orig=x, asucc=succ(x.a)\}$

Has type $\{a:Nat\} \rightarrow \{orig:\{a:Nat\}, asucc:Nat\}$

fpoly = λX . λx :X. x

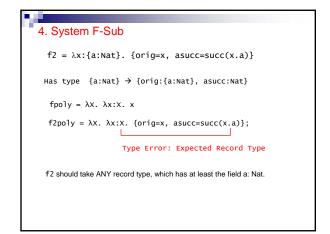
f2poly = λx . λx : {orig=x, asucc=succ(x.a)};

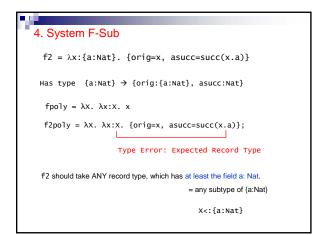
4. System F-Sub

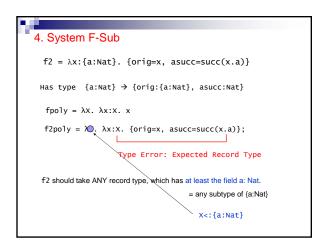
f2 = $\lambda x: \{a: Nat\}$. {orig=x, asucc=succ(x.a)} Has type {a:Nat} \rightarrow {orig:{a:Nat}, asucc:Nat} fpoly = λx . $\lambda x: x$. x

f2poly = λx . λx :x. {orig=x, asucc=succ(x.a)};

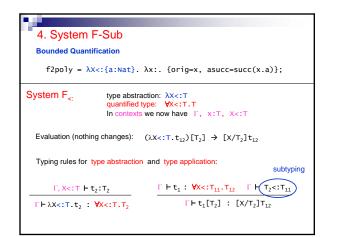
Type Error: Expected Record Type

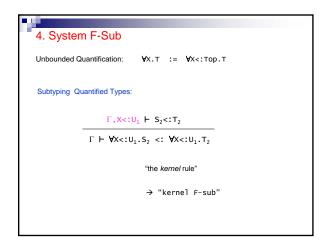


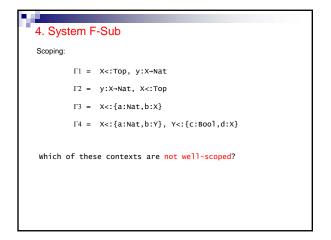


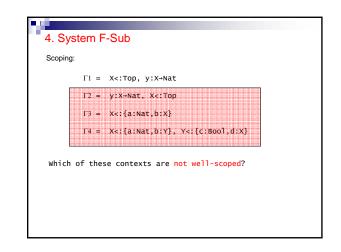


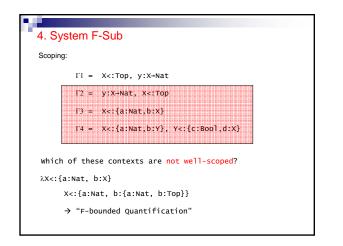
Bounded Quantification	
f2poly = $\lambda X <: \{a:Nat\}$.	<pre>λx:. {orig=x, asucc=succ(x.a)};</pre>
quantified	action: λX<:T type: ∀X<:T.T we now have Γ, x:T, X<:T
Evaluation (nothing changes):	$(\lambda X \leq :T.t_{12})[T_2] \rightarrow [X/T_2]t_{12}$
Typing rules for type abstraction	and type application:
Γ , X<:T \vdash t ₂ :T ₂	$\Gamma \vdash t_1 : \forall X <: T_{11} \cdot T_{12} \Gamma \vdash T_2 <: T_{11}$
Γ⊢λX<:T.t, : ∀X<:T.T,	FHt ₁ [T ₂] : [X/T ₂]T ₁₂

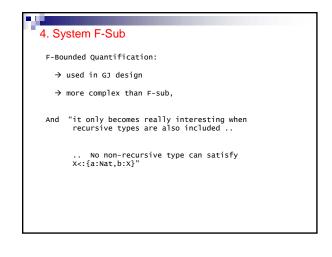


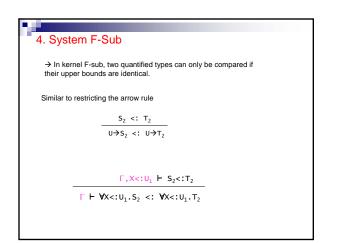


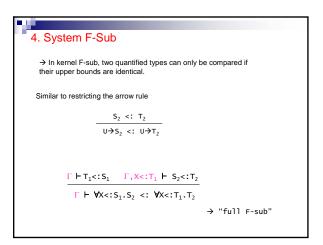












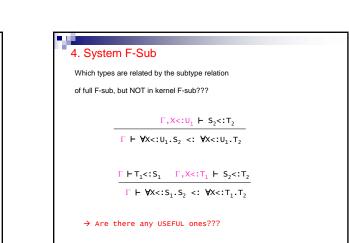
4. System F-Sub

Which types are related by the subtype relation of full F-sub, but NOT in kernel F-sub???

 $\Gamma, X <: U_1 \vdash S_2 <: T_2$

 $\Gamma \vdash \forall X <: U_1.S_2 <: \forall X <: U_1.T_2$

 $\frac{\Gamma \vdash \mathsf{T}_1 <: \mathsf{S}_1 \qquad \Gamma, \mathsf{X} <: \mathsf{T}_1 \vdash \mathsf{S}_2 <: \mathsf{T}_2}{\Gamma \vdash \forall \mathsf{X} <: \mathsf{S}_1 . \mathsf{S}_2 <: \forall \mathsf{X} <: \mathsf{T}_1 . \mathsf{T}_2}$



5. Properties of F-Sub

Preservation If $t \vdash t:T$ and $t \rightarrow t'$, then $\Gamma \vdash t':T$.

Progress

If t is a closed and well-typed term, then either

t is a value, or t \rightarrow t' for some term t'.

Proofs: Induction on the structure of terms.

→Use canonical forms lemma: If v is closed value of type $T_1 \rightarrow T_2$, then $v = \lambda x : S_1 \cdot t_2$

If v is closed value of type $\forall X <: T_1.T_2$, then $v = \lambda X <: T_1.t_2$.

5. Properties of F-Sub Theorem. Typing and Subtyping in kernel F-sub is decidable. Theorem. Subtyping in full F-sub is undecidable. Next time: (1) prove these theorems. (2) look at FGJ = FJ+generics.