Where are we

• So far: Added subsumption and subtyping rules

\[ \Gamma \vdash e : \tau_1 \quad \tau_1 \tau_2 \]

\[ \Gamma \vdash e : \tau_2 \]

• Immutable records: width, permutation, depth
  – Depth = covariant fields
• Functions: contravariant argument, covariant result
• Transitive and reflexive

• And… this subtyping has no run-time effect!
  – Tempting to go beyond: coercions & downcasts
Coercions

Some temptations

1. int float “numeric conversion”
2. int {l1= int} “autoboxing”
3. τ string “implicit marshalling / printing”
4. τ1 τ2 “overload the cast operator”

These all require run-time actions for subsumption – called coercions

Keeps programmers from whining about float_of_int and obj.toString(), but…
Coherence problems

• Now program behavior can depend on:
  – “where” subsumption occurs in type-checking
  – “how” $\tau_1 \quad \tau_2$ is derived

• These are called “coherence” problems

Two “how” examples:

1. print_string(34) where int float and $\tau$ string
   • Can “fix” by printing ints with trailing .0

2. 34==34 where int $\{l1=\text{int}\}$ and == is bit-equality for all types
Languages with “incoherent” subtyping must define
  – Where subsumption occurs
  – What the derivation order is

Typically complicated and incomplete (or arbitrary)

C++ example (Java interfaces similar, unsure about C#)

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

• A separate issue: downcasts
• Easy to explain a checked downcast:

\[
\text{if\_hastype}(\tau, e1) \quad \text{then} \quad x \rightarrow e2 \quad \text{else} \quad e3
\]

“Roughly, if at run-time \( e1 \) has type \( \tau \) (or a subtype), then bind it to \( x \) and evaluate \( e2 \). Else evaluate \( e3 \).”

• Just to show the issue is orthogonal to exceptions

• In Java you use \textit{instanceof} and a cast
Bad results

Downcasts exist and help avoid limitations of incomplete type systems, but they have drawbacks:

1. (The obvious:) They can fail at run-time
2. Types don’t erase (need tags; ML doesn’t)
3. Breaks abstractions: without them, you can pass \{l1=1,l2=2\} and \{l1=1,l2=3\} to f : \{l1=int\}->int and know you get the same answer!
4. Often a quick workaround when you should use parametric polymorphism…
Our plan

• Simply-typed Lambda-Calculus
• Safety = (preservation + progress)
• Extensions (pairs, datatypes, recursion, etc.)
• Digression: static vs. dynamic typing
• Digression: Curry-Howard Isomorphism
• Subtyping
• Type Variables:
  – Generics (\(\forall\)), Abstract types (\(\exists\)), (Recursive types)
• Type inference
The goal

Understand this interface 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 specification
2. Client: Use code meeting this specification
What the client likes

1. Library is reusable
   • Different lists with elements of different types
   • New reusable functions outside 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

3. Library behaves the same for all type instantiations!
   – e.g.:
     ```
     length (cons 3 mt_list)
     length (cons 4 mt_list)
     length (cons (7,9) mt_list)
     ```
   must be totally equivalent
   – In theory, less (re)-integration testing
What the library likes

1. Reusability
   • For same reasons as clients

2. Abstraction of $\text{mylist}$ from clients
   • Clients behave the same for all equivalent implementations
     – e.g.: can change from tree list to array
   • Clients typechecked knowing only there exists a type constructor $\text{mylist}$
   • Clients cannot cast a $\tau$ $\text{mylist}$ to its hidden implementation
For simplicity…

Our example has a lot going on:
1. Element types held *abstract* from library
2. Type constructor held *abstract* from client
3. Type-variable reuse expresses type-equalities
4. Recursive types for data structures

Will mostly focus on (1) and (3)
   – then (2) (without type constructors)
   – Just a minute or two on (4)

Our theory will differ from ML (explain later)

Much more interesting than “doesn’t get stuck”
Syntax

New:
- Type variables and *universal types*
- Contexts include “what type variables in scope”
- Explicit type *abstraction* and *instantiation*

\[
e ::= c \mid x \mid \lambda x: \tau. e \mid e \ e \mid \Lambda \alpha. e \mid e [\tau]
\]

\[
v ::= \lambda x: \tau. e \mid c \mid \Lambda \alpha. e
\]

\[
\tau ::= \text{int} \mid \tau \to \tau \mid \alpha \mid \forall \alpha. \tau
\]

\[
\Gamma ::= \cdot \mid \Gamma, x: \tau \mid \Gamma, \alpha
\]
Semantics

- Left-to-right small-step CBV needs only 2 new rules:

\[ e \rightarrow e' \]
\[ e[\tau] \rightarrow e'[\tau] \]
\[ (\Lambda \alpha. e)[\tau] \rightarrow e\{\tau/\alpha\} \]

- But: must also define \( e\{\tau/\alpha\} \) (and \( \tau'\{\tau/\alpha\} \))
  - Much like \( e\{v/x\} \) (including capture issues)
  - \( \Lambda \) and \( \forall \) are both bindings (can shadow)
- e.g., (using +): \[ (\Lambda \alpha. \Lambda \beta. \lambda x:\alpha. \lambda f:\alpha\rightarrow\beta. f \ x) \]
  \[ [\text{int}] \ [\text{int}] \ 3 \ (\lambda y:\text{int}.y+y) \]
Typing

- Mostly just be picky: no free type variables ever
- Let $\Gamma \vdash \tau$ mean all free type variables are in $\Gamma$
  - Rules straightforward and important but boring
- 2 new rules (and 1 picky new premise on old rule)

\[
\Gamma, \alpha \vdash e : \tau \quad \Gamma \vdash e : \forall \alpha. \tau_1 \quad \Gamma \vdash \tau_2
\]

\[
\Gamma \vdash (\Lambda \alpha. e) : \forall \alpha. \tau
\]

\[
\Gamma \vdash e [\tau_2] : \tau_1\{\tau_2/\alpha}\]

- e.g.:

\[
(\Lambda \alpha. \Lambda \beta. \lambda x: \alpha. \lambda f: \alpha \rightarrow \beta. f \ x)
\]

\[
[int] \ [int] \ 3 \ (\lambda y: \text{int}. y+y)
\]
The Whole Language (System F)

\[ e ::= c \mid x \mid \lambda x : \tau. \ e \mid e \ e \mid \Lambda \alpha. \ e \mid e [\tau] \]

\[ v ::= \lambda x : \tau. \ e \mid c \mid \Lambda \alpha. \ e \]

\[ \tau ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \forall \alpha. \ \tau \]

\[ \Gamma ::= . \mid \Gamma, x : \tau \mid \Gamma, \alpha \]

\[ e_1 \rightarrow e_1' \]
\[ e_2 \rightarrow e_2' \]
\[ e \rightarrow e' \]

---

\[ (\lambda x. e) \ v \rightarrow e\{v/x\} \]

\[ (\Lambda \alpha. \ e) [\tau] \rightarrow e\{\tau/\alpha\} \]

---

\[ \Gamma, x : \tau_1 \vdash e : \tau_2 \quad \Gamma \vdash \tau_1 \rightarrow \tau_2 \quad \Gamma \vdash e_2 : \tau_1 \]

\[ \Gamma, \alpha \vdash e : \tau \]

\[ \Gamma, (\Lambda \alpha. \ e) : \forall \alpha. \tau \]

\[ \Gamma \vdash e [\tau_2] : \tau_1{\tau_2/\alpha} \]

\[ \Gamma \vdash c : \text{int} \]
Examples with id

Polymorphic identity: let $id = (\forall \alpha. \lambda x:\alpha . x)$

- $id$ has type $\forall \alpha . \alpha \rightarrow \alpha$
- $id$ [int] has type int$\rightarrow$int
- $id$ [int*int] has type (int*int)$\rightarrow$(int*int)
- $(id$ $[\forall \alpha . \alpha \rightarrow \alpha ]$ id) has type $\forall \alpha . \alpha \rightarrow \alpha$

In ML, you cannot do the last two; in System F you can.
More examples

- Let applyOld = $(\forall \alpha. \forall \beta. \lambda x: \alpha. \lambda f: \alpha \rightarrow \beta. \ f \ x)$
  - Type $\forall \alpha. \ \forall \beta. \ \alpha \rightarrow (\alpha \rightarrow \beta) \rightarrow \beta$
- Let applyNew = $(\forall \alpha. \ \lambda x: \alpha. \forall \beta. \lambda f: \alpha \rightarrow \beta. \ f \ x)$
  - Type $\forall \alpha. \ \alpha \rightarrow (\forall \beta. (\alpha \rightarrow \beta) \rightarrow \beta)$
  - Impossible in ML
  - Interesting when using “partial application”
- Let twice = $(\forall \alpha. \ \lambda x: \alpha. \lambda f: \alpha \rightarrow \alpha. \ f \ (f \ x))$
  - Type $\forall \alpha. \ \alpha \rightarrow (\alpha \rightarrow \alpha) \rightarrow \alpha$
  - Cannot be made more polymorphic
    - argument and result types must agree
Facts

Amazing-but-true things we won’t prove:
1. **Type-safe** (preservation + progress)
2. All programs **terminate**
   - shocking: we saw self-application
   - so add let rec
3. **Parametricity** (a.k.a. “theorems for free”)
   - Example: If $e : \forall \alpha. \forall \beta. (\alpha \ast \beta) \rightarrow (\beta \ast \alpha)$, then $e$ is the swap function, i.e., equivalent to $(\Lambda \alpha. \Lambda \beta. \lambda x : \alpha \ast \beta. (x \cdot 2, x \cdot 1))$
4. **Erasure** (no need for types at run-time)
Where are we

Understanding parametric polymorphism
- Define System F, a core model of universal types
- Saw simple examples
- Stated some surprising theorems

Now:
- A parametricity for “security” example
- Relate to ML
  - Less powerful than System F
  - But easier to infer types and “usually enough”
Security from safety?

Example: A thread shouldn’t access files it didn’t open
• Even if passed a file-handle from another thread
• That way fopen can check a thread’s permissions

The rough set-up:
• Typecheck an untrusted thread-body e:
  \[ e : \forall \alpha. \{ \text{fopen}=\text{string} \rightarrow \alpha, \text{fread}=\alpha \rightarrow \text{int} \} \rightarrow \text{unit} \]
• Type-check spawn:
  \[ \text{spawn}: (\forall \alpha. \{ \text{fopen}=\text{string} \rightarrow \alpha, \text{fread}=\alpha \rightarrow \text{int} \} \rightarrow \text{unit}) \rightarrow \text{unit} \]
• Implement spawn \( v \):
  “enqueue” (\( v [\text{int}] \{ \text{fopen}=\lambda x: \text{string}..., \text{fread}=\lambda x: \text{int}...\} \))
What happened

In our example:

• At run-time, file-handles are just ints
• But type-checker is told threads are polymorphic over file-handle types
  – So “maybe” spawn uses a different type for each thread’s file handles
  – So “it typechecks” ensures no passing file-handles
  – So we can check permissions at fopen instead of fread (more secure and faster)

In general: memory safety is necessary but insufficient for language-based enforcement of abstractions
Connection to reality

• System F has been one of the most important theoretical PL models since the early 70s and inspires languages like ML

• But ML-style polymorphism “feels different”
  1. It is implicitly typed
  2. It is more restrictive
  3. (1) and (2) have everything to do with each other
ML in terms of System F

Four restrictions:
1. All types look like $\forall \alpha_1 \ldots \forall \alpha_n. \tau$ where $n \geq 0$ and $\tau$ has no $\forall$ characters.
   - Called “prenex-quantification” or “lack of first-class polymorphism”

2. Only let (rec) variables (e.g., $x$ in `let x=e1 in e2`) can have polymorphic types.
   - i.e., $n=0$ for function arguments, anonymous functions, pattern variables, etc.
   - Called “let-bound polymorphism”
   - So cannot “desugar let to lambda” in ML
ML in terms of System F

Four restrictions:

3. For \texttt{let rec} \( f \ x = e_1 \) where \( f \) has type \( \forall \alpha_1 \ldots \forall \alpha_n. \tau_1 \rightarrow \tau_2 \), every use of \( f \) in \( e_1 \) looks like \( f \ [\alpha_1] \ldots [\alpha_n] \) (after inference)
   - Called “no polymorphic recursion”

4. Let variables can be polymorphic only if the expression bound to them is a “syntactic value”
   - Functions, constructors, and variables allowed
   - Applications are not
   - (Prevents unsoundness due to mutation)
   - Called “the value restriction”
Why?

- These restrictions are usually tolerable.
- Polymorphic recursion makes inference undecidable
  - Proven in 1992
- 1st-class polymorphism makes inference undecidable
  - Proven in 1995
- Note: Type inference for ML *efficient* in practice, but not in theory: A program of size $n$ and run-time $n$ can have a type of size $O(2^{2^n})$
- The value restriction is one way to prevent an unsoundness with references
Going beyond

“Good” extensions to ML still being considered. A case study for “what matters” for an extension:

- **Soundness:** Does the system still have its “nice properties”?
- **Conservatism:** Does the system still typecheck every program it used to?
- **Power:** Does the system typecheck “a lot” of new programs?
- **Convenience:** Does the system not require “too many” explicit annotations.
Where are we

- System F explains type abstraction, generics, etc.
  - Universal types ($\forall \alpha. \tau$) use type variables ($\alpha$)
- Now: two other uses of type variables
  - Recursive types (will largely skip)
  - Existential types
- Need both for defining/implementing our interface

```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
```
Recursive types

• To define recursive data structures, you need some sort of recursive type.

• Most PLs, including ML use named types, e.g.,

\[
\text{type 'a mylist = Empty | Cons of 'a * ('a list)}
\]

(* issue is orthogonal to type-constructors: *)

\[
\text{type intlist = Empty | Cons of int * (int list)}
\]

• Just like “fix” is “nameless” unlike letrec, “mu-types” are nameless at the type level
  – \( \mu \alpha. \tau \) is “equal to its unrolling” \( \tau \{ \mu \alpha. \tau / \alpha \} \)
  – “intlist” can be \( \mu \alpha. (\text{unit} + (\text{int} * \alpha)) \)
Key facts we’re skipping

1. In a subtyping world:
   – Want subtyping between recursive types, e.g.:

   ```
   type t1 = A | B of int * t1 | C of t1 * t1
   type t2 = A | B of int * t2
   (* sound: t2 <= t1 *)
   ```

   – subtyping judgment needs a context
   – efficient algorithm & its soundness non-obvious (early 90s)

2. STLC + mu-types Turing complete
   • Translating untyped lambda into it actually easy

3. Practical impact example: XML Schema matching
Toward abstract types

```
type intlist (* simpler: no type constructor! *)
val mt_list : intlist
val cons : int -> intlist -> intlist
val decons : intlist -> ((int * intlist) option)
val length : intlist -> int
val map : (int -> int) -> intlist -> intlist
```

• Abstraction of library from clients: definition of myintlist
  – In ML, use the (second-class) module system
• Try to encode this abstraction 3 ways
  – Universal types (awkward and backward)
  – OO idiom (the “binary method problem”)
  – Existential types (the right new thing)
• We’ll stay informal
Approach #1

Use universal types like we did for file-handles:

$$(\Lambda \beta. \lambda x: \tau_1. \text{client}) \:[\tau_2] \: \text{library}$$

Where:

- $\tau_1 = \{ \text{mt\_list} = \beta, \newline\text{cons} = \text{int} \rightarrow \beta \rightarrow \beta, \newline\text{decons} = \beta \rightarrow ((\text{int}\times\beta)\text{option}), \newline... \}$
- $\tau_2$ is an implementation (e.g., type $t = C \mid D$ of int*t)
- Client uses record projection to get functions
Less than ideal

\[(λβ.  λ x : τ_1 . \text{client}) [τ_2] \text{library}\]

Plus:
- It works without new language features

Minus:
- The “wrong side” is saying “keep this abstract”
- Different list-libraries have different types
  - “2nd-class” – cannot make a table of them or choose the one you want at run-time
  - Fixing it requires a “structure inversion” – choosing a library and passing a client to it (impractical for large programs)
Approach #2

The “OO” approach – use an “object” implementing an interface for each library

(* the interface *)

\[
\text{type } t = \{ \text{cons : int -> } t; \\
\text{decons : unit -> } ((\text{int} \times t) \text{ option}); \\
\text{length : unit -> int } \}
\]

(*library #1, "methods" use "fields" i and l *)

let rec cons_f i l =
    let rec r = \{ cons = (fun j -> cons_f j r); \\
    decons = (fun() -> Some (i,l)); \\
    length = (fun() -> 1 + l.length ()) \} \\
    in r

let mt_list1 =
    let rec r = \{ cons = (fun j -> cons_f j r); \\
    decons = (fun() -> None); \\
    length = (fun() -> 0) \} \\
    in r
Approach #2 continued

(* the interface (repeated to remind you) *)

\begin{verbatim}
type t = { cons : int -> t;
           decons : unit -> ((int * t) option);
           length : unit -> int }
\end{verbatim}

(*library #2, "methods" use "fields" l and len *)

\begin{verbatim}
let rec make i l =
  let len = List.length l in
  { cons = (fun j -> make (j::l));
    decons = (fun() -> match l with
                |[]     -> None
                |hd::tl -> Some(hd,make tl));
    length = (fun() -> len)}
let mt_list2 = make []
\end{verbatim}
Approach #2 continued

Now have two “libraries” for making values of type t
• First has faster constructor, decons, slower length
• Second has slower constructor, decons, faster length
• This is very OO (more than most code in OO PLs?!)  
  Whole point: same type means first-class  
    – We “forget” which library made it

```ocaml
let lst : t list = [mt_list1;
                  mt_list2;
                  mt_list1.cons(7);
                  mt_list2.cons(9)]
```
Less than ideal

• Plus
  – First-class libraries are easy
  – Fits in OOP world (nothing new)

• Minus
  – “Nobody” can knows what library an object is from
  – Fine except for “binary” (really n\geq2) methods

• “Weak” example:
  – **Must** implement append using cons and decons!

```haskell
type t = {cons : int -> t;
          decons : unit -> ((int * t) option);
          length : unit -> int;
          append : t -> t }
```
Strong binary methods

• In previous example, libraries couldn’t “optimize” append, but at least append could be implemented

• “Strong” example:

```ocaml
type t = { cons : int -> t;
            average : unit -> int;
            append : t -> t
            }
```

• Can implement values of this type, but you cannot have them do what you want!

• In practice, widen interfaces (which is exactly what abstraction is supposed to avoid)
Approach #3

- “use what we already have” approaches had minuses
- Direct way to model abstract types are existential types: $\exists \alpha . \tau$
- Can be formalized, but we’ll just
  - Show the idea
  - Show how you can use it to encode closures

• Sermon: Existential types have been understood for 20 years; they belong in our PLs
Our library with

\begin{verbatim}
\texttt{pack} \texttt{list\_type, \texttt{list\_as}}
\end{verbatim}

\begin{itemize}
  \item A universal is definitely wrong here
  \item Another lib would \texttt{pack} different type and code but
  \begin{verbatim}
  \begin{array}{l}
  \texttt{mt\_list = } \beta, \\
  \texttt{cons } = \texttt{int } \rightarrow \beta \rightarrow \beta, \\
  \texttt{decons } = \texttt{(int}\ast\beta)\texttt{option}, \\
  \end{array}
  \end{verbatim}
\end{itemize}

- Binary methods no problem:
  \begin{itemize}
  \item Uses must \texttt{unpack} and different unpacks will have
    \begin{verbatim}
    \texttt{\beta, append = } \phi \rightarrow \phi \rightarrow \phi
    \end{verbatim}
  \item have the same type (so can put in a list)
  \end{itemize}

\begin{verbatim}
...\end{verbatim}
Closures & Existentials

- There’s a deep connection between \( \exists \) and how closures are (1) used and (2) compiled
- “Call-backs” are the canonical example:

```ocaml
(* interface *)
val onKeyEvent : (int->unit)->unit

(* implementation *)
let callbacks : (int->unit) list ref = ref []
let onKeyEvent f =
    callbacks := f::(!callbacks)
let keyPress i =
    List.iter (fun f -> f i) !callbacks
```
The connection

- Key to flexibility:
  - Each registered callback can have “private fields” of different types
  - But each callback has type int->unit
- In C, we don’t have closures or existentials, so we use void* (next slide)
  - Clients must downcast their environment
  - Clients must assume library passes back correct environment
/* interface */
typedef struct{void* env; void(*f)(void*,int);}* cb_t;
void onKeyEvent(cb_t);

/* implementation (assuming a list library) */
list_t callbacks = NULL;
void onKeyEvent(cb_t cb){
    callbacks=cons(cb, callbacks);
}
void keyPress(int i) {
    for(list_t lst=callbacks; lst; lst=lst->tl)
        lst->hd->f(lst->hd->env, i);
}
The type we want

- The `cb_t` type should be an existential:
  ```c
  /* interface using existentials (not C) */
  typedef struct{∃α. α env; void(*f)(α, int);}* cb_t;
  void onKeyEvent(cb_t);
  ```

- Client does a “pack” to make the argument for `onKeyEvent`
  - Must “show” the types match up
- Library does an “unpack” in the loop
  - Has no choice but to pass each `cb_t` function pointer its own environment
- See Cyclone if curious (syntax ain’t pretty; concept is)
Our plan

- Simply-typed Lambda-Calculus
- Safety = (preservation + progress)
- Extensions (pairs, datatypes, recursion, etc.)
- Digression: static vs. dynamic typing
- Digression: Curry-Howard Isomorphism
- Subtyping
- Type Variables:
  - Generics ($\forall$), Abstract types ($\exists$), (Recursive types)
- Type inference

Note: Bounded polymorphism next time (cf. hw4 #2)
The ML type system

Have already defined (most of) the ML type system
• System F with 4 restrictions
• (Plus bells, whistles, and a module system)
• (No subtyping or overloading)

Semi-revisionist history; this type system is what a simple, elegant *inference algorithm* supports
• Called “Algorithm W” or “Hindley-Milner inference”
• In theory, inference “fills out explicit types”
• In practice, often merge inference and checking

An algorithm best understood by example…
Example #1

```ml
let f x =  
  let (y,z) = x in  
  (abs y) + z
```
Example #2

```ocaml
let rec sum lst =  
    match lst with  
      [] -> 0  
    | hd::tl -> hd + (sum tl)
```
Example #3

```ocaml
let rec length lst =  
  match lst with  
  | [] -> 0  
  | hd::tl -> 1 + (length tl)
```
Example #4

```
let compose f g x = f (g x)
```
More generally

- Infer each let-binding or toplevel binding in order
  - Except for mutual recursion (do all at once)
- Give each variable and subexpression a fresh “constraint variable”
- Add constraints for each subexpression
  - Very similar to typing rules
- Circular constraints fail (so $x \ x$ never typechecks)
- After inferring let-body, *generalize* (turn unconstrained constraint variables into type variables)
In practice

• Described algorithm as
  – “generate a ton of constraints”
  – “solve them” (stop on failure, generalize at let)
• In practice, faster “unification-based” algorithm is equivalent:
  – Each constraint variable is a “pointer” (a reference)
  – Equality constraints become indirection
  – Can shorten paths eagerly
• Value restriction done separately (see need next time)