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)
- Type inference

Polymorphism

- Key source of restrictiveness in our types so far: Given a \( \Gamma \), there is at most one \( \tau \) such that \( \Gamma \vdash e : \tau \)
- Various forms of polymorphism allow more terms to type-check
  - Ad hoc: \( e1+e2 \) in SML < C < Java < C++
  - Parametric: "generics" \( 'a \to 'a \) can also have type \( \text{int}\to\text{int} \), \( ('b\to'b)\to('b\to'b) \), etc.
  - Subtype: new Vector().add(new C()) is legal pre-generics Java because new C() can have type Object because C \leq Object
- Try to avoid the ambiguous word "polymorphism"
  - Prefer "static overloading", "dynamic dispatch", "type abstraction", "subtyping"

How to add subtyping

Key idea: A value of subtype should "make sense" (not lead to stickiness) wherever a value of supertype is expected
  - Hence what is a subtype is, "not a matter of opinion"

Capture key idea with just one new typing rule (for \( \Gamma \vdash e : \tau \))
  - Leaving all the action to a new "helper" judgment \( \tau_1 \leq \tau_2 \)

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

To see a language with [more] interesting subtyping opportunities we'll add records to our typed lambda-calculus...
Width

This doesn’t yet type-check but it’s safe:

(* f : (l1=int, l2=int)-> int *)

let f = λx. x.l1 + x.l2 in

(f {l1=3, l2=4})

+ (f {l1=7, l2=8, l3=9})

• f has to have one type, but wider arguments okay

• Suggests a first inference rule for our new τ₁ ≤ τ₂ judgment:

{ l1=τ₁, ..., ln=τₙ } ≤ { l₁=τ₁, ..., ln=τₙ }

– Allows 1 new field, but can use the rule multiple times

Transitivity

• To derive . | {19=7,122=4,10=λx. x.l1 } : {19=int}

we could use subsumption twice with our width rule each time

• But it’s more convenient and sensible to be able to derive

{19=int,122=int,10=λl1=int)->int} ≤ {19=int}

• In general, can accomplish this with a transitivity rule for our subtyping judgment

τ₁ ≤ τ₂  τ₂ ≤ τ₃

⇒ τ₁ ≤ τ₃

– Now a type-checker can at each point use subsumption at most once, asking a helper function, “I have a τ₁ and need a τ₂; am I cool?”

Permutation

• Why should field order in the type matter?

  – For safety, it doesn’t

• So this permutation rule is sound:

  – Again transitivity makes this enough

{l₁=τ₁, ..., l₁=τ₁, j₁=τ₂, ..., ln=τₙ}

≤ {l₁=τ₁, ..., j₁=τ₁, l₁=τ₂, ..., ln=τₙ}

• Note in passing: Efficient algorithms to decide if τ₁ ≤ τ₂ are not always simple or existent

  – Not hard with rules shown so far

Digression: Efficiency

• With our semantics, width and permutation make perfect sense

• But many type systems restrict one or both to make fast compilation easier

Goals:

1. Compile x.l₁ to memory load at known offset
2. Allow width subtyping
3. Allow permutation subtyping
4. Compile record values without (many) "gaps"

All 4 impossible in general, any 3 is pretty easy

  – Metapoint: Type systems often have restrictions motivated by compilers, not semantics

Toward depth

Recall we added width to type-check this code:

let f = λx. x.l1 + x.l2 in

(f {l1=3, l2=4})

+ (f {l1=7, l2=8, l3=9})

But we still can’t type-check this code:

let f = λx. x.l1.l1 + x.l1.l2 in

(f {l1={l1=3, l2=4}})

+ (f {l1={l1=7, l2=8, l3=9}})

Want subtyping “deeper” in record types…

Depth

• This rule suffices

τ₁ ≤ τ

{l₁=τ₁, ..., l₁=τ₁, ln=τₙ}

≤ {l₁=τ₁, ..., l₁=τ₁, ln=τₙ}

• A height n derivation allows subtyping n levels deep

• But is it sound?

  – Yes, but only because fields are immutable!!

  – Once again a restriction adds power elsewhere!

  – Why is immutability key for this rule? See also: HW4
Toward function subtyping

- So far allow some record types where others expected
- What about allowing some function types where others expected
- For example,
  \[
  \text{int} \rightarrow \{ l1=\text{int}, l2=\text{int} \} \leq \text{int} \rightarrow \{ l1=\text{int} \}
  \]
- But what's the general principle?

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

Function subtyping

\[
\tau_3 \leq \tau_1 \quad \tau_2 \leq \tau_4 \quad \text{Also want:} \quad \tau \leq \tau
\]

- Supertype can impose more restrictions on arguments and reveal less about results
- Jargon: Contravariant in argument, covariant in result
- Example:
  \[
  \{ l1=\text{int}, l2=\text{int} \} \rightarrow \{ l1=\text{int}, l2=\text{int} \} \leq \{ l1=\text{int}, l2=\text{int}, l3=\text{int} \} \rightarrow \{ l1=\text{int} \}
  \]

Let me be clear

- Functions are contravariant in their argument and covariant in their result
- Similarly, in class-based OOP, an overriding method could have contravariant argument types and covariant result type
  - But many languages aren't so useful
- Covariant argument types are wrong!!!
  - Please remember this
  - For safety, but see Dart and Typescript and Eiffel
  - Can “method missing error” occur at run-time?

Where are we

- So far: Added subsumption and subtyping rules
  \[
  \Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2
  \]
  \[
  \Gamma \vdash e : \tau_2
  \]
  - And... this subtyping has no run-time effect!
  - Tempting to go beyond: coercions & downcasts

Summary

\[
\begin{align*}
\Gamma \vdash e : \tau_1 \quad \tau_1 \leq \tau_2 & \quad \Gamma \vdash e : \tau_2 \quad \tau \leq \tau_3 \\
\tau_1 \rightarrow \tau_2 \leq \tau_3 \rightarrow \tau_4 & \quad \tau_3 \leq \tau_1 \quad \tau_2 \leq \tau_4
\end{align*}
\]

Coercions

Some temptations
1. \( \text{int} \leq \text{float} \) “numeric conversion”
2. \( \text{int} \leq \{ l1=\text{int} \} \) “autoboxing”
3. \( \tau \leq \text{string} \) “implicit marshalling / printing”
4. \( \tau_1 \leq \tau_2 \) “overload the cast operator”

These all require run-time actions for subsumption
- Called coercions

Keeps programmers from whining © about \texttt{float.of_int} and \texttt{obj.toString()}, but...
Coherence problems

- Now program behavior can depend on:
  - "where" subsumption occurs in type-checking
  - "how" $\tau_1 \leq \tau_2$ is derived
- These are called "coherence" problems

Two "how" examples:
- `print_string(34)` where `int \leq float` and `\tau \leq string`
  - Can "fix" by printing ints with trailing .0
- `34==34` where `int \leq \{l1=int\}` and `==` is bit-equality

It's a mess

Languages with "incoherent" subtyping must define
- Where subsumption occurs
- What the derivation order is
Typically complicated, incomplete and/or arbitrary

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

```cpp
class C2 {};
class C3 {};
class C1 : public C2, public C3 {};
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:
  ```
  if_hastype(\tau, e1) then x -> e2 else 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 `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 run-time tags where ML doesn't
3. Breaks abstractions: without them, you can pass `{ll=1,l2=2}` and `{ll=1,l2=3}` to $f : \{ll=int\} \rightarrow \text{int}$ 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)
- Type inference

The goal

Understand this interface and why it matters:

```typescript
type 'a mylist
val empty_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 empty_list) 
     length (cons 4 empty_list) 
     length (cons (7,9) empty_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 `mylist` from clients
   - Clients behave the same for all equivalent implementations
     - e.g.: can change to an `arrayList`
   - Clients typechecked knowing only there exists a type constructor `mylist`
   - Clients cannot cast a `τ mylist` to its hidden implementation

   Allowing programmers to define their own abstractions is an essential obligation (??) of a PL

What now?

So to understand the essential ideas of type variables, we could extend our formal typed lambda calculus with:

- Types like ∀α. ∀β. α→(α→β)→β
- Functions that take types as well as values (generics)
- Type constructors (take types to produce types)
- Modules with abstract types

Instead we'll use pseudocode

- Reminiscent of OCaml
- But this is not code that works in OCaml
- Will then explain why OCaml is actually more restrictive
  - (It's for type inference)

Basics

- Let functions take types as well as values
  - Made up syntax
  - Still just currying
    ```
    (*map: ∀a. ∀b. (a->b)->a list->b list*)
    ```
    - In body, type variables are in scope just like term variables
      - Use for calling other polymorphic functions
    ```
    let map <int> <bool> (fun x->x=2) [1;2;3]
    ```

Basics, cont'd

- Instantiating a type variable does substitution
  - Just like calling a function with a value does
  - So `map<int>` would be
    ```
    <b> fun (f:int->'b) -> fun (lst:int list) ->
    match lst with
    [ ] -> [ ]
    | hd::tl -> (::<b>) (f hd) (map<int><'b> f tl)
    ```
  - In types or programs, can consistently rename type variables
    - So these are two ways to write the same type
      ```
      ∀'a. ∀'b. ('a->'b)->'a list->'b list
      ∀'foo. ∀'bar. ('foo->'bar)->'foo list->'bar list
      ```

What can you do with types?

- The only thing we "do" with types is instantiate generic functions
  - And all callees "do" with type arguments is other instantiations
  - So these types have no run-time effect
    - That is, a pre-pass could erase them all
    - That is, an interpreter/compiler can ignore them

- This "erasure" property doesn't hold if allow run-time type operations like `instanceof` or C#-style dynamic dispatch
  - Or C++-style overloading
  - These break abstraction...
Abstraction

Without type operations, callee cannot branch on (i.e., "tell") how its type arguments were instantiated

- That is why foo<int> [1;2;3] and foo<int*int> [(1,4);(2,5);(3,6)] must return the same value
- And why any function of type \( \forall a \forall b. (\langle a^*a \rangle \rightarrow \langle b^*b \rangle) \)
  swaps its arguments or raises an exception or diverges or...
  - Its behavior does not depend on the argument
- This is "parametricity" a.k.a. "theorems for free"
  - Type variables enforce strong abstractions

Fancier stuff

- As defined, our pseudocode (but not OCaml) allows:
  - First-class polymorphism
  - Polymorphic recursion

- First-class polymorphism: can pass around/return functions that take type variables
  - Example using currying:
    ```
    let prependAll<'a>(x:'a)<'b>(lst:'b list) =
    map <'a> <'b> (fun y -> (x,y)) lst
    let addZeros = prependAll <int> 0
    ```

Polymorphic recursion

```ocaml
let rec funnyCount<'a><'b>(f:'a->bool) (g:'b->bool)(lst1:'a list)(lst2:'b list) =
  match lst1 with
  | [] -> 0 (* weird, lst2 might not be empty *)
  | hd::tl -> (if (f hd) then 1 else 0)
              + funnyCount<'a><'b>(g f lst2)(h prime)

let useFunny = funnyCount <int> <bool> (fun x -> x=4) not [2;4;4] [true;false]
```

Onto abstract types

That’s enough about universal types for now
  - Inference and combining with subtyping later

Now what about the other part of our example
  - A signature shows a module defines a type (or type constructor) but not its implementation
  - A slightly simpler example:
    ```
    type intSet
    val single : int -> intSet
    val union : intSet -> intSet -> intSet
    ```

Why abstraction

There are an infinite number of equivalent implementations
  - With different trade-offs, e.g., performance

- Example: fast union but no duplicate removal
  ```
  type intSet = S of int | U of intSet * intSet
  let union sl sls2 = U(s1,s2)
  ```

- Example: fast lookup for 42, no other duplicate removal
  ```
  type intSet = bool * (int list)
  let single i = if i=42 then (true,[i])
               else (false,[i])
  ```

```ocaml
let useFunny = funnyCount <int> <bool> (fun x -> x=4) not [2;4;4] [true;false]
```
The backwards E

What does our interface "say" to clients of it

```ocaml
type intSet
val single : int -> intSet
val contains : intSet -> bool
val union : intSet -> intSet -> intSet
```

"There exists" a type, call it intSet, such that these values have these types

This is not the same thing as, "For all" alpha, foo can take an alpha

To confuse "for all" vs. "there exists" is like confusing "and" vs. "or"

- not (p1 and p2) == (not p1) or (not p2)
- not (exists a. (not p)) == forall a. p

Versus OOP

OOP types also have a "there exists" aspect to them with this/self hiding the implementation via private fields

- May study OOP later
- "Binary methods" (e.g., union) don't quite work out cleanly!
  - Without downcasts or other "cheats"

```csharp
// still non-imperative (orthogonal issue)
interface IntSet {
    boolean contains(int);
    IntSet union(IntSet);
}
```

The key difference

- In OCaml, the implementation of union "knew" the underlying representation of its arguments
- On the other hand, if OCaml has two different libraries, they have different types, so you can't choose one at run-time
- It is possible to have first-class abstract types
  - Also known as existential types
  - Show the basic idea in a different domain: closures in C
- Demonstrates the lower-level implementation of OCaml closures is related to "there exists"

Closures & Existentials

- There's a deep connection between 3 and how closures are (1) used and (2) compiled
- "Call-backs" are the canonical example:

```
(* 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
Now in C

/* 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);
}

/* clients: full of casts to/from void* */

The type we want

• The cb_t type should be an existential:

/* interface using existentials (not C) */
typedef
struct{\alpha. \alpha env; void(*f)(\alpha, 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
• This is not a forall
  • (I played around with this stuff to get my Ph.D. 😊 and now see
    Rust and such…)

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)
• Type inference

Where are we

• Done: understand subtyping
• Done: understand “universal” types and “existential” types
• Now: Bounded parametric polymorphism
  – Synergistic combination of universal and subtyping
• Then: making universal types easier to use but less powerful
  – Type inference
  – Reconsider first-class polymorphism / polymorphic recursion
  – Polymorphic-reference problem
• Then done (??) with types

Why bounded polymorphism

Could one language have \( \tau_1 \leq \tau_2 \) and \( \forall \alpha. \tau \) ?
  – Sure! They’re both useful and complementary
  – But how do they interact?

1. When is \( \forall \alpha. \tau_1 \leq \forall \beta. \tau_2 \) ?

2. What about bounds?

   let dbl1 l x = x.l1 <- x.l1*2; x
   – Subtyping: dbl1 : \{l1=int\} \rightarrow \{l1=int\}
     • Can pass subtype, but result type loses a lot
   – Polymorphism: dbl1 : \forall \alpha. \alpha \rightarrow \alpha
     • Lose nothing, but body doesn’t type-check

What bounded polymorphism

The type we want: \( \text{dbl1} : \forall \alpha (l1\text{int}). \alpha \rightarrow \alpha \)

Java and C# generics have this (different syntax)

Key ideas:

• A bounded polymorphic function can use subsumption as
  specified by the constraint
• Instantiating a bounded polymorphic function must satisfy the
  constraint
Subtyping revisited

When is $\alpha \leq \tau_1 \leq \tau_2 \leq \tau_3 \leq \tau_4$?

- Note: already “alpha-converted” to same type variable

Sound answer:
- Contravariant bounds ($\tau_3 \leq \tau_1$)
- Covariant bodies ($\tau_2 \leq \tau_4$)

Problem: Makes subtyping undecidable (1992; surprised many)

Common workarounds:
- Require invariant bounds ($\tau_3 \leq \tau_1$ and $\tau_1 \leq \tau_3$)
- Some ad hoc approximation

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 ($\exists$)
- Type inference

The ML type system

- Called “Algorithm W” or “Hindley-Milner inference”
- In theory, inference “fills out explicit types”
  - Complete if finds an explicit typing whenever one exists
- In practice, often merge inference and checking

An algorithm best understood by example…
- Then we’ll explain the type system for which it actually infers types
- Yes, this is backwards: how does it do it, before defining it

Example #1

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

Example #2

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

Example #3

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

```ocaml
let compose f g x = f (g x)
```

Example #5

```ocaml
let rec funnyCount f g lst1 lst2 =
  match lst1 with
  | [] -> 0 (* weird, lst2 might not be empty *)
  | hd::tl -> (if (f hd) then 1 else 0)
    + funnyCount g f lst2 tl

(* does not type-check: *)
let useFunny =
  funnyCount (fun x -> x=4) not
  [2;4;4] [true;false] *
```

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-expression, generalize (unconstrained constraint variables become type variables)

Note: Actual implementations much more efficient than “generate big pile of constraints then solve” (can unify eagerly)

What this infers

“Natural” limitations of this algorithm: Universal types, but
1. Only let-bound variables get polymorphic types
   - This is why let is not sugar for fun in OCaml
2. No first-class polymorphism (all foralls all the way to the left)
3. No polymorphic recursion

Unnatural limitation imposed for soundness reasons we will see:
4. “Value restriction”: let x = e1 in e2 gives x a polymorphic type only if e1 is a value or a variable
   - Includes e1 being a function
   - OCaml has relaxed this slightly in some cases

Why?

- These restrictions are usually tolerable
- Polymorphic recursion makes inference undecidable
  - Proven in 1992
- First-class polymorphism makes inference undecidable
  - Proven in 1995
- Note: Type inference for OCaml 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

Given this…

Subject to these 4 limitations, inference is perfect:

- It gives every expression the most general type it possibly can
  - Not all type systems even have most-general types
- So every program that can type-check can be inferred
  - That is, explicit type annotations are never necessary
  - Exceptions are related to the “value restriction”
    - Make programmer specify non-polymorphic type
Polymorphic references

A sound type system cannot accept this program:

```plaintext
let x = ref [] in
x := 1::[];
match !x with _ -> () | hd::_ -> hd ^ "gotcha"
```

But it would assume this interface:

```plaintext
type 'a ref
val ref : 'a -> 'a ref
val ! : 'a ref -> 'a
val := : 'a ref -> 'a -> unit
```

Solutions

Must restrict the type system.

Many ways exist

1. “Value restriction”: \ref {} cannot have a polymorphic type
   - syntactic look for \ref not enough
2. Let \ref {} have type \((\forall \alpha. \alpha \text{ list}) \text{ ref}\)
   - not useful and not an ML type
3. Tell the type system “mutation is special”
   - not “just another library interface”

Going beyond

What makes a “good extension” to a type system?

- **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?