CSEP505: Programming Languages
Lecture 7: Subtypes, Type Variables

Dan Grossman
Autumn 2016

STLC in one slide

Expressions: $e ::= x \mid \lambda x.\ e \mid e\ e \mid c$
Values: $v ::= \lambda x.\ e \mid c$
Types: $\tau ::= int \mid \tau \to \tau$
Contexts: $\Gamma ::= . \mid \Gamma, x:\tau$

$e \to e'$

$$\frac{e1 \to e1'}{e1\ e2 \to e1'\ e2} \qquad \frac{e2 \to e2'}{v\ e2 \to v\ e2'} \qquad \frac{}{(\lambda x.e)\ v \to e\{v/x\}}$$

$\Gamma \vdash e : \tau$

$$\frac{}{\Gamma \vdash c : int} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma, x:\tau 1 \vdash e : \tau 2}{\Gamma \vdash (\lambda x.e) : \tau 1 \to \tau 2} \qquad \frac{\Gamma \vdash e1 : \tau 1 \to \tau 2 \quad \Gamma \vdash e2 : \tau 1}{\Gamma \vdash e1\ e2 : \tau 2}$$

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 (∀), Abstract types (∃)
- Type inference

Polymorphism

- Key source of restrictiveness in our types so far:
  Given a Γ, there is at most one τ 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 -> 'a` can also have type `int->int`, `('b->'b)->('b->'b)`, etc.
  - *Subtype*: `new Vector().add(new C())` is legal pre-generics Java because `new C()` can have type `Object` because $C \le$ `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 stuckness) 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 $\tau 1 \le \tau 2$

$$\frac{\Gamma \vdash e : \tau 1 \quad \tau 1 \le \tau 2}{\Gamma \vdash e : \tau 2}$$

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

Records w/o polymorphism

Like pairs, but fields named and any number of them:
Field names: l (distinct from variables)
Exps: $e ::= \ldots \mid \{l_1=e, \ldots, l_n=e\} \mid e.l$
Types: $\tau ::= \ldots \mid \{l_1=\tau, \ldots, l_n=\tau\}$

$$\frac{e \to e'}{\{l_1=v1, \ldots, l_i=vi, l_j=e, \ldots, l_n=en\} \to \{l_1=v1, \ldots, l_i=vi, l_j=e', \ldots, l_n=en\}} \qquad \frac{e \to e'}{e.l \to e'.l}$$

$$\frac{}{\{l_1=v1,\ldots,l_i=vi,\ldots,l_n=vn\}.\ l_i \to vi} \qquad \frac{\Gamma \vdash e : \{l_1=\tau 1,\ldots,l_n=\tau n\}}{\Gamma \vdash e.\ l_i : \tau i}$$

$$\frac{\Gamma \vdash e1 : \tau 1 \ \ldots\ \Gamma \vdash en : \tau n \quad \text{"labels distinct"}}{\Gamma \vdash \{l_1=e1, \ldots, l_n=en\} : \{l_1=\tau 1,\ldots,l_n=\tau n\}}$$

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 $\tau 1 \leq \tau 2$ judgment:

$$\frac{}{\{l1{=}\tau 1,\ldots, ln{=}\tau n,\ l{=}\tau\} \leq \{l1{=}\tau 1,\ldots, ln{=}\tau n\}}$$

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

Transitivity

- To derive $\cdot \vdash$ `{l9=7,l22=4,l0=λx. x.l1 } : {l9=int}` we could use subsumption twice with our width rule each time
- But it's more convenient and sensible to be able to derive `{l9=int,l22=int,l0={l1=int}->int}` $\leq$ `{l9=int}`
- In general, can accomplish this with a *transitivity rule* for our subtyping judgment

$$\frac{\tau 1 \leq \tau 2 \quad \tau 2 \leq \tau 3}{\tau 1 \leq \tau 3}$$

  - Now a type-checker can at each point use subsumption at most once, asking a helper function, "I have a $\tau 1$ and need a $\tau 2$; 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

$$\frac{}{\{l1{=}\tau 1,\ \ldots,\ li{=}\tau i, lj{=}\tau j,\ \ldots,\ ln{=}\tau n\} \leq \{l1{=}\tau 1,\ \ldots,\ lj{=}\tau j, li{=}\tau i,\ \ldots,\ ln{=}\tau n\}}$$

- Note in passing: Efficient *algorithms* to decide if $\tau 1 \leq \tau 2$ 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

$$\frac{\tau i \leq \tau}{\{l1{=}\tau 1,\ldots, li{=}\tau i,\ldots, ln{=}\tau n\} \leq \{l1{=}\tau 1,\ldots, li{=}\tau,\ldots, ln{=}\tau n\}}$$

- 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,
  $\texttt{int} \to \{l1\texttt{=int}, l2\texttt{=int}\} \le \texttt{int} \to \{l1\texttt{=int}\}$
- But what's the general principle?

$$\frac{??????}{\tau 1 \to \tau 2 \le \tau 3 \to \tau 4}$$

Function subtyping

$$\frac{\tau 3 \le \tau 1 \quad \tau 2 \le \tau 4}{\tau 1 \to \tau 2 \le \tau 3 \to \tau 4}$$

Also want: $\frac{}{\tau \le \tau}$

- Supertype can impose more restrictions on arguments and reveal less about results
- Jargon: *Contravariant* in argument, *covariant* in result
- Example:
  $\{l1\text{= int}, l2\text{= int}\} \to \{l1\text{= int}, l2\text{= int}\}$
  $\le \{l1\text{= int}, l2\text{= int}, l3\text{= int}\} \to \{l1\text{= int}\}$

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?

Summary

$$\frac{\Gamma \vdash e : \tau 1 \quad \tau 1 \le \tau 2}{\Gamma \vdash e : \tau 2} \qquad \frac{}{\tau \le \tau} \qquad \frac{\tau 1 \le \tau 2 \quad \tau 2 \le \tau 3}{\tau 1 \le \tau 3} \qquad \frac{\tau 3 \le \tau 1 \quad \tau 2 \le \tau 4}{\tau 1 \to \tau 2 \le \tau 3 \to \tau 4}$$

$$\frac{}{\{l1=\tau 1, \ldots, ln=\tau n, \; l=\tau\} \le \{l1=\tau 1, \ldots, ln=\tau n\}}$$

$$\frac{}{\{l1=\tau 1, \ldots, li=\tau i, lj=\tau j, \ldots, ln=\tau n\} \le \{l1=\tau 1, \ldots, lj=\tau j, li=\tau i, \ldots, ln=\tau n\}}$$

$$\frac{\tau i \le \tau}{\{l1=\tau 1, \ldots, li=\tau i, \ldots, ln=\tau n\} \le \{l1=\tau 1, \ldots, li=\tau, \ldots, ln=\tau n\}}$$

Where are we

- So far: Added subsumption and subtyping rules

$$\frac{\Gamma \vdash e : \tau 1 \quad \tau 1 \le \tau 2}{\Gamma \vdash e : \tau 2}$$

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

Coercions

Some temptations

1. $\texttt{int} \le \texttt{float}$ "numeric conversion"
2. $\texttt{int} \le \{l1\texttt{=int}\}$ "autoboxing"
3. $\tau \le \texttt{string}$ "implicit marshalling / printing"
4. $\tau 1 \le \tau 2$ "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" `τ1 ≤ τ2` is derived
- These are called "coherence" problems

Two "how" examples:

- `print_string(34)` where `int ≤ float` and `τ ≤ string`
  - Can "fix" by printing ints with trailing `.0`
- `34==34` where `int ≤ {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#)

```
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:

```
if_hastype(τ,e1) then x -> e2 else e3
```

Roughly, "if at run-time e1 has type `τ` (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 `{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 (∀), Abstract types (∃)
- Type inference

The goal

Understand this interface and why it matters:

```
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 $\forall\alpha.\ \forall\beta.\ \alpha \to (\alpha \to \beta) \to \beta$
- 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)*)
let map <'a> <'b> f lst = …
```

- In body, type variables are in scope just like term variables
  – Use for calling other polymorphic functions

```
let ftf = map <int> <bool> (fun x->x=2) [1;2;3]

let map <'a> <'b> (f:'a->'b) (lst:'a list) =
    match lst with
       [] -> []
     | hd::tl -> (::<'b>) (f hd)
                          (map<'a><'b> f tl)
```


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

  `∀'a ∀'b. ('a*'b)->('b*'a)`

  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

(* ∀'b. 'b list -> (int * 'b) list *)
let addZeros = prependAll <int> 0
```


Fancier stuff

- Polymorphic recursion: A polymorphic function can call itself using a different instantiation
- Silly example:

```
let f <'a> (g:'a->bool) (x:'a) (i:int)=
   if g x
   then f<int> (fun x -> x % 2 = 0) i i
   else f<int*int> (fun p -> true) (3,4) 2
```

- Useful (??) example…

Polymorphic recursion

```
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 <'b> <'a> g f lst2 tl

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 contains : intSet -> int -> bool
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 s1 s2 = 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,[])
                       else (false,[i])
…
let union(b1,lst1)(b2,lst2) = (b1||b2, lst1@lst2)
```

The backwards E

What does our interface "say" to *clients* of it

```
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 "forall" 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"

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


Versus OOP

```
interface IntSet {
  boolean contains(int);
  IntSet union(IntSet);
}
class MyIntSet implements IntSet {
  private boolean has42 = false;
  private IntList lst = null;
  MyIntSet(int x) { … }
  boolean contains(int x) { … }
  IntSet union(IntSet that) { /* Good luck! */ }
}
```

Cannot do all of:

1. Write `MyIntSet` "how we want"
2. Have `MyIntSet` implement `IntSet` (w/o changing `IntSet`)
3. Have `union` return a `MyIntSet`
4. Not insert casts and failures

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 ∃ 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{∃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
- 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 (∀), Abstract types (∃)
- 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 dblL1 x = x.l1 <- x.l1*2; x
```

   - Subtyping: `dblL1 : {l1=int} → {l1=int}`
     - Can pass subtype, but result type loses a lot
   - Polymorphism: `dblL1 : ∀α.α → α`
     - Lose nothing, but body doesn't type-check

What bounded polymorphism

The type we want: `dblL1 : ∀α≤{l1=int}.α→α`

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 $\forall\alpha\leq\tau1.\tau2 \leq \forall\alpha\leq\tau3.\tau4$?

- Note: already "alpha-converted" to same type variable

Sound answer:

- Contravariant bounds ($\tau3\leq\tau1$)
- Covariant bodies ($\tau2\leq\tau4$)

Problem: Makes subtyping undecidable (1992; surprised many)

Common workarounds:

- Require invaraint bounds ($\tau3\leq\tau1$ and $\tau1\leq\tau3$)
- 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 (∀), Abstract types (∃)
- 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

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


Example #2

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


Example #3

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

Example #4

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


Example #5

```
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:

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

But it would assuming this interface:

```
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 `(∀α.α list) 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?