Programming Languages

Type Inference, Subtyping & equivalence

Jiwon Seo

Type-checking

- (Static) type-checking can reject a program before it runs to prevent the possibility of some errors
  - A feature of statically typed languages

- Dynamically typed languages do little (none?) such checking
  - So might try to treat a number as a function at run-time

- Will study relative advantages after some Racket
  - Racket (and Python, Ruby, Javascript, …) dynamically typed

- ML (and Java, C#, Scala, C, C++) is statically typed
  - Every binding has one type, determined “at compile-time”

Implicitly typed

- ML is statically typed
- ML is implicitly typed: rarely need to write down types

```
fun f x = (* infer val f : int -> int *)
    if x > 3
    then 42
    else x * 2

fun g x = (* report type error *)
    if x > 3
    then true
    else x * 2
```

- Statically typed: Much more like Java than Javascript!

Type inference

- Type inference problem: Give every binding/expression a type such that type-checking succeeds
  - Fail if and only if no solution exists
- In principle, could be a pass before the type-checker
  - But often implemented together
- Type inference can be easy, difficult, or *impossible*
  - Easy: Accept all programs
  - Easy: Reject all programs
  - Subtle, elegant, and *not magic*: ML

Overview

- Will describe ML type inference via several examples
  - General algorithm is a slightly more advanced topic
  - Supporting nested functions also a bit more advanced

- Enough to help you "do type inference in your head"
  - And appreciate it is not magic

Key steps

- Determine types of bindings in order
  - (Except for mutual recursion)
  - So you cannot use later bindings: will not type-check
- For each **`val`** or **`fun`** binding:
  - Analyze definition for all necessary facts (constraints)
  - Example: If see **`x > 0`**, then **`x`** must have type **`int`**
  - Type error if no way for all facts to hold (over-constrained)
- Afterward, use type variables (e.g., **`'a`**) for any unconstrained types
  - Example: An unused argument can have any type
- (Finally, enforce the *value restriction*, discussed later)

Very simple example

After this example, will go much more step-by-step

- Like the automated algorithm does

```
val x = 42

fun f (y, z, w) =
    if y
    then z + x
    else 0
(* f :  *)
```

Very simple example

After this example, will go much more step-by-step

- Like the automated algorithm does

```
val x = 42 (* val x : int *)

fun f (y, z, w) =
    if y (* y must be bool *)
    then z + x (* z must be int *)
    else 0 (* both branches have same type *)
(* f must return an int
   f must take a bool * int * ANYTHING
   so val f : bool * int * 'a -> int
 *)
```

Relation to Polymorphism

- Central feature of ML type inference: it can infer types with type variables
  - Great for code reuse and understanding functions
- But remember there are two orthogonal concepts
  - Languages can have type inference without type variables
  - Languages can have type variables without type inference

Key Idea

- Collect all the facts needed for type-checking

- These facts constrain the type of the function

- See code examples for:
  - Two examples without type variables
  - And one example that does not type-check
  - Then examples for polymorphic functions
    - Nothing changes, just under-constrained: some types can “be anything” but may still need to be the same as other types

Two more topics

- ML type-inference story so far is too lenient
  - Value restriction limits where polymorphic types can occur
  - See why and then what

- ML is in a “sweet spot”
  - Type inference more difficult without polymorphism (generic type)
  - Type inference more difficult with subtyping

Important to “finish the story” but these topics are:

- A bit more advanced
- A bit less elegant

The Problem

As presented so far, the ML type system is *unsound*!

- Allows putting a value of type **`t1`** (e.g., **`int`**) where we expect a value of type **`t2`** (e.g., **`string`**)

A combination of polymorphism and mutation is to blame:

```
val r = ref NONE (* val r : 'a option ref *)

r := SOME "hi"

val i = 1 + valOf (!r)
```

- Assignment type-checks because (infix) **`:=`** has type **`'a ref * 'a -> unit`**, so instantiate with **`string option`**
- Dereference type-checks because **`!`** has type **`'a ref -> 'a`**, so instantiate with **`int option`**

What to do

To restore soundness, need a stricter type system that rejects at least one of these three lines

```
val r = ref NONE (* val r : 'a option ref *)

r := SOME "hi"

val i = 1 + valOf (!r)
```

- And cannot make special rules for reference types because type-checker cannot know the definition of all type synonyms
  - Module system

```
type 'a foo = 'a ref
val f = ref (* val f : 'a -> 'a foo *)
val r = f NONE
```

The fix

```
val r = ref NONE (* val r : ?.X1 option ref *)

r := SOME "hi"

val i = 1 + valOf (!r)
```

- Value restriction: a variable-binding can have a polymorphic type only if the expression is a variable or value
  - Function calls like **`ref NONE`** are neither

- Else get a warning and unconstrained types are filled in with dummy types (basically unusable)

- Not obvious this suffices to make type system sound, but it does

The downside

As we saw previously, the value restriction can cause problems when it is unnecessary because we are not using mutation

```
val pairWithOne = List.map (fn x => (x,1))
(* does not get type 'a list -> ('a*int) list *)
```

The type-checker does not know **`List.map`** is not making a mutable reference

Saw workarounds in previous segment on partial application

- Common one: wrap in a function binding

```
fun pairWithOne xs = List.map (fn x => (x,1)) xs
(* 'a list -> ('a*int) list *)
```

A local optimum (good trade-off)

- Despite the value restriction, ML type inference is elegant and fairly easy to understand
- More difficult *without* polymorpism (generic type)
  - What type should length-of-list have?
- More difficult *with* subtyping
  - Suppose pairs are supertypes of wider tuples
  - Then **`val (y,z) = x`** constrains **`x`** to have at least two fields, not exactly two fields
  - Depending on details, languages can support this, but types often more difficult to infer and understand

Generic Types and Subtypes

- Generics
  - In ML, generic types are types with type variables
    - `'a`, `'a` list, `'c` option , `'a` → `'b` …
  - In Java Generics, in C++ Templates

- Subtypes
  - Not in ML
  - In Java, C#, C++, use subclass to implement subtype
  - But must satisfy: Liskov substitution principle

    `S` <: **`T`** then any value of type **`S`** must be usable in every way a **`T`** is

Classes vs. Types

- A class defines an object's behavior
  - Subclassing inherits behavior and changes it via overriding

- A type describes an object's methods’ argument/result types
  - A subtype is substitutable in terms of its field/method types

- These are separate concepts: try to use the terms correctly
  - Java/C# confuse them by requiring subclasses to be subtypes
  - A class name is both a class and a type
  - Confusion is convenient in practice

Classes vs. Types

- A class defines an object's behavior
  - Subclassing inherits behavior and changes it via overriding

- A type describes an object's methods' argument/result types
  - A subtype is substitutable in terms of its field/method types

- These are separate concepts: try to use the terms correctly
  - Java/C# confuse them by requiring subclasses to be subtypes
  - A class name is both a class and a type
  - Confusion is convenient in practice

Subtyping in a tiny language

- Can cover most core subtyping ideas by just considering *records with mutable fields*

- Will make up our own syntax
  - ML has records, but no subtyping or field-mutation
  - Java uses class/interface names and rarely fits on a slide

Records (half like ML, half like Java)

Record creation (field names and contents):

`{f1=e1, f2=e2, …, fn=en}` Evaluate **`ei`**, make a record

Record field access:

`e.f` Evaluate **`e`** to record **`v`** with an **`f`** field, get contents of **`f`** field

Record field update

`e1.f = e2` Evaluate **`e1`** to a record **`v1`** and **`e2`** to a value **`v2`**; Change **`v1`**'s **`f`** field (which must exist) to **`v2`**; Return **`v2`**

A Basic Type System

Record types: What fields a record has and type for each field

`{f1:t1, f2:t2, …, fn:tn}`

Type-checking expressions:

- If **`e1`** has type **`t1`**, …, **`en`** has type **`tn`**,
  then **`{f1=e1, …, fn=en}`** has type **`{f1:t1, …, fn:tn}`**
- If **`e`** has a record type containing **`f : t`**,
  then **`e.f`** has type **`t`**
- If **`e1`** has a record type containing **`f : t`** and **`e2`** has type **`t`**,
  then **`e1.f = e2`** has type **`t`**

This is safe

These evaluation rules and typing rules prevent ever trying to access a field of a record that does not exist

Example program that type-checks (in a made-up language):

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val pythag : {x:real,y:real} = {x=3.0, y=4.0}
val five : real = distToOrigin(pythag)
```

Motivating subtyping

But according to our typing rules, this program does not type-check

– It does nothing wrong and seems worth supporting

```
fun distToOrigin (p:{x:real,y:real}) =
  Math.sqrt(p.x*p.x + p.y*p.y)

val c : {x:real,y:real,color:string} =
   {x=3.0, y=4.0, color="green"}

val five : real = distToOrigin(c)
```

A good idea: allow extra fields

Natural idea: If an expression has type

`{f1:t1, f2:t2, …, fn:tn}`

Then it can *also* have a type with some fields removed

This is what we need to type-check these function calls:

```
fun distToOrigin (p:{x:real,y:real}) = …
fun makePurple (p:{color:string}) =
    p.color = "purple"

val c :{x:real,y:real,color:string} =
   {x=3.0, y=4.0, color="green"}

val _ = distToOrigin(c)
val _ = makePurple(c)
```

Keeping subtyping separate

A programming language already has a lot of typing rules and we do not want to change them

- Example: The type of an actual function argument must ***equal*** the type of the function parameter

We can do this by adding "just two things to our language"

- *Subtyping*: Write **`t1 <: t2`** for **`t1`** is a subtype of **`t2`**
- One new typing rule that uses subtyping:

  If **`e`** has type **`t1`** and **`t1 <: t2`**,

  then **`e`** (also) has type **`t2`**

Now all we need to do is define **`t1 <: t2`**

Subtyping rule

- Principle of *substitutability*: If **`t1 <: t2`**, then any value of type **`t1`** must be usable in every way a **`t2`** is (== whenever t2 value is used, we should replace it with t1 value)
  - Here: Any value of subtype needs all fields any value of supertype has

Four good rules

For our record types, these rules all meet the substitutability test:

1. "Width" subtyping: A supertype can have a subset of fields with the same types
2. "Permutation" subtyping: A supertype can have the same set of fields with the same types in a different order
3. Transitivity: If `t1 <: t2` and `t2 <: t3`, then `t1 <: t3`
4. Reflexivity: Every type is a subtype of itself

More record subtyping?

[Warning: I am misleading you ☺]

Subtyping rules so far let us drop fields but not change their types

Example: A circle has a center field holding another record

```
fun circleY (c:{center:{x:real,y:real}, r:real}) =
   c.center.y

val sphere:{center:{x:real,y:real,z:real}, r:real} =
  {center={x=3.0,y=4.0,z=0.0}, r=1.0}

val _ = circleY(sphere)
```

For this to type-check, we need:

```
{center:{x:real,y:real,z:real}, r:real}
                  <:
     {center:{x:real,y:real}, r:real}
```

Do not have this subtyping – could we?

```
{center:{x:real,y:real,z:real}, r:real}
                  <:
     {center:{x:real,y:real}, r:real}
```

- No way to get this yet: we can drop **`center`**, drop **`r`**, or permute order, but cannot "reach into a field type" to do subtyping
- So why not add another subtyping rule… "Depth" subtyping:
  If **`ta <: tb`**, then **`{f1:t1, …, f:ta, …, fn:tn} <:`**
  `{f1:t1, …, f:tb, …, fn:tn}`
- Depth subtyping (along with width on the field's type) lets our example type-check

Stop!

- It is nice and all that our new subtyping rule lets our example type-check
- But it is not worth it if it breaks soundness
  - Also allows programs that can access missing record fields
- Unfortunately, it breaks soundness ☹

Mutation strikes again

If **`ta <: tb`**,
then **`{f1:t1, …, f:ta, …, fn:tn} <:`**
`{f1:t1, …, f:tb, …, fn:tn}`

```
fun setToOrigin (c:{center:{x:real,y:real}, r:real})=
   c.center = {x=0.0, y=0.0}

val sphere:{center:{x:real,y:real,z:real}, r:real} =
  {center={x=3.0, y=4.0, z=0.0}, r=1.0}

val _ = setToOrigin(sphere)
val _ = sphere.center.z (* kaboom! (no z field) *)
```

Moral of the story

- In a language with records/objects with getters and setters, depth subtyping is unsound
  - Subtyping cannot change the type of fields
- If fields are immutable, then depth subtyping is sound!
  - Yet another benefit of outlawing mutation!
  - Choose two of three: setters, depth subtyping, soundness
- Remember: subtyping is not a matter of opinion

Picking on Java (and C#)

Arrays should work just like records in terms of depth subtyping

- But in Java, if `t1 <: t2`, then `t1[] <: t2[]`
- So this code type-checks, surprisingly

```
class Point { … }
class ColorPoint extends Point { … }
…
void m1(Point[] pt_arr) {
  pt_arr[0] = new Point(3,4);
}
String m2(int x) {
  ColorPoint[] cpt_arr = new ColorPoint[x];
  for(int i=0; i < x; i++)
     cpt_arr[i] = new ColorPoint(0,0,"green");
  m1(cpt_arr); // !
  return cpt_arr[0].color; // !
}
```

Now functions

- Already know a caller can use subtyping for arguments passed
  - Or on the result

- More interesting: When is one function type a subtype of another?

  - Important for higher-order functions: If a function expects an argument of type **`t1 -> t2`**, can you pass a **`t3 -> t4`** instead?

  - Coming next: Important for understanding methods
    - (An object type is a lot like a record type where “method positions” are immutable and have function types)

Example

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flip p = {x = ~p.x, y=~p.y}
val d = distMoved(flip, {x=3.0, y=4.0})
```

No subtyping here yet:

- **`flip`** has exactly the type **`distMoved`** expects for **`f`**
- Can pass **`distMoved`** a record with extra fields for **`p`**, but that's old news

Return-type subtyping

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipGreen p = {x = ~p.x, y=~p.y, color="green"}
val d = distMoved(flipGreen, {x=3.0, y=4.0})
```

- Return type of **`flipGreen`** is **`{x:real,y:real,color:string}`**, but **`distMoved`** expects a return type of **`{x:real,y:real}`**
- Nothing goes wrong: If **`ta <: tb`**, then **`t->ta <: t->tb`**
  - A function can return "*more* than it needs to"
  - Jargon: "Return types are *covariant*"

This is wrong

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipIfGreen p = if p.color = "green" (*kaboom!*)
                    then {x = ~p.x, y=~p.y}
                    else {x = p.x, y=p.y}
val d = distMoved(flipIfGreen, {x=3.0, y=4.0})
```

- Argument type of **`flipIfGreen`** is **`{x:real,y:real,color:string}`**, but it is called with a **`{x:real,y:real}`**
- Unsound! **`ta <: tb`** does **NOT** allow **`ta -> t <: tb -> t`**

The other way works!

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipX_Y0 p = {x = ~p.x, y=0.0}
val d = distMoved(flipX_Y0, {x=3.0, y=4.0})
```

- Argument type of **`flipX_Y0`** is **`{x:real}`**, but it is called with a **`{x:real,y:real}`**, which is fine
- If **`tb <: ta`**, then **`ta -> t <: tb -> t`**
  - A function can assume "*less* than it needs to" about arguments
  - Jargon: "Argument types are *contravariant*"

Can do both

```
fun distMoved (f : {x:real,y:real}->{x:real,y:real},
               p : {x:real,y:real}) =
   let val p2 : {x:real,y:real} = f p
       val dx : real = p2.x - p.x
       val dy : real = p2.y - p.y
   in Math.sqrt(dx*dx + dy*dy) end

fun flipXMakeGreen p = {x = ~p.x, y=0.0, color="green"}
val d = distMoved(flipXMakeGreen, {x=3.0, y=4.0})
```

- **`flipXMakeGreen`** has type

  `{x:real} -> {x:real,y:real,color:string}`
- Fine to pass a function of such a type as function of type

  `{x:real,y:real} -> {x:real,y:real}`
- If **`t3 <: t1`** and **`t2 <: t4`**, then **`t1 -> t2 <: t3 -> t4`**

Conclusion

- If `t3 <: t1` and `t2 <: t4`, then `t1 -> t2 <: t3 -> t4`
  - Function subtyping contravariant in argument(s) and covariant in results
- Also essential for understanding subtyping and methods in OOP
- Most unintuitive concept in the course
  - Smart people often forget and convince themselves covariant arguments are okay
  - These people are always mistaken

Last Topic – Equivalence

More careful look at what “two pieces of code are equivalent” means

- Fundamental software-engineering idea

- Made easier with
  - Abstraction (hiding things)
  - Fewer side effects

Not about any “new ways to code something up”

Equivalence

Must reason about “are these equivalent” *all the time*

- The more precisely you think about it the better

- *Code maintenance:* Can I simplify this code?
- *Backward compatibility:* Can I add new features without changing how any old features work?
- *Optimization:* Can I make this code faster?
- *Abstraction:* Can an external client tell I made this change?

To focus discussion: When can we say two functions are equivalent, even without looking at all calls to them?

- May not know all the calls (e.g., we are editing a library)

A definition

Two functions are equivalent if they have the same “observable behavior” no matter how they are used anywhere in any program

Given equivalent arguments, they:

- Produce equivalent results
- Have the same (non-)termination behavior
- Mutate (non-local) memory in the same way
- Do the same input/output
- Raise the same exceptions

Notice it is much easier to be equivalent if:

- There are fewer possible arguments, e.g., with a type system and abstraction
- We avoid *side-effects*: mutation, input/output, and exceptions

Example

Since looking up variables in ML has no side effects, these two functions are equivalent:

```
fun f x = x + x
```

$=$

```
val y = 2
fun f x = y * x
```

But these next two are not equivalent in general: it depends on what is passed for **`f`**

– Are equivalent *if* argument for **`f`** has no side-effects

```
fun g (f,x) =
    (f x) + (f x)
```

$\neq$

```
val y = 2
fun g (f,x) =
    y * (f x)
```

– Example: `g ((fn i => print "hi" ; i), 7)`

– Great reason for "pure" functional programming

Another example

Are these equivalent? Or not equivalent?

```
fun f x =
    let
      val y = g x
      val z = h x
    in
      (y,z)
    end
```

$\stackrel{?}{=}$

```
fun f x =
    let
      val z = h x
      val y = g x
    in
      (y,z)
    end
```

Another example

These are equivalent *only if* functions bound to **`g`** and **`h`** do not raise exceptions or have side effects (printing, updating state, etc.)

– Again: pure functions make more things equivalent

```
fun f x =
    let
      val y = g x
      val z = h x
    in
      (y,z)
    end
```

$\neq$

```
fun f x =
    let
      val z = h x
      val y = g x
    in
      (y,z)
    end
```

– Example: **`g`** divides by **`0`** and **`h`** mutates a top-level reference

– Example: **`g`** writes to a reference that **`h`** reads from

Syntactic sugar

Using or not using syntactic sugar is always equivalent

– By definition, else not syntactic sugar

Example:

```
fun f x =
    x andalso g x
```

$=$

```
fun f x =
    if x
    then g x
    else false
```

But be careful about evaluation order

```
fun f x =
    x andalso g x
```

$\neq$

```
fun f x =
    if g x
    then x
    else false
```

Standard equivalences

Three general equivalences that always work for functions

- In any (?) decent language

1. Consistently rename bound variables and uses

```
val y = 14
fun f x = x+y+x
```

$=$

```
val y = 14
fun f z = z+y+z
```

But notice you can’t use a variable name already used in the function body to refer to something else

```
val y = 14
fun f x = x+y+x
```

$\neq$

```
val y = 14
fun f y = y+y+y
```

```
fun f x =
   let val y = 3
   in x+y end
```

$\neq$

```
fun f y =
   let val y = 3
   in y+y end
```

Standard equivalences

Three general equivalences that always work for functions

- In (any?) decent language

2. Use a helper function or do not

```
val y = 14
fun g z = (z+y+z)+z
```

=

```
val y = 14
fun f x = x+y+x
fun g z = (f z)+z
```

But notice you need to be careful about environments

```
val y = 14
val y = 7
fun g z = (z+y+z)+z
```

≠

```
val y = 14
fun f x = x+y+x
val y = 7
fun g z = (f z)+z
```

Standard equivalences

Three general equivalences that always work for functions

- In (any?) decent language

3. Unnecessary function wrapping

```
fun f x = x+x
fun g y = f y
```

$=$

```
fun f x = x+x
val g = f
```

But notice that if you compute the function to call and *that computation* has side-effects, you have to be careful

```
fun f x = x+x
fun h () = (print "hi";
            f)
fun g y = (h()) y
```

$\neq$

```
fun f x = x+x
fun h () = (print "hi";
            f)
val g = (h())
```

One more

If we ignore types, then ML let-bindings can be syntactic sugar for calling an anonymous function:

```
let val x = e1
in e2 end
```

```
(fn x => e2) e1
```

- These both evaluate **`e1`** to **`v1`**, then evaluate **`e2`** in an environment extended to map **`x`** to **`v1`**
- So *exactly* the same evaluation of expressions and result

But in ML, there is a type-system difference:

- **`x`** on the left can have a polymorphic type, but not on the right
- Can always go from right to left
- If **`x`** need not be polymorphic, can go from left to right

What about performance?

According to our definition of equivalence, these two functions are equivalent, but we learned one is awful

- (Actually we studied this before pattern-matching)

```
fun max xs =
  case xs of
    [] => raise Empty
  | x::[] => x
  | x::xs' =>
      if x > max xs'
      then x
      else max xs'
```

```
fun max xs =
  case xs of
    [] => raise Empty
  | x::[] => x
  | x::xs' =>
     let
       val y = max xs'
     in
       if x > y
       then x
       else y
     end
```

Different definitions for different jobs

- PL Equivalence: given same inputs, same outputs and effects
  - Good: Lets us replace bad **`max`** with good **`max`**
  - Bad: Ignores performance in the extreme
- Asymptotic equivalence: Ignore constant factors
  - Good: Focus on the algorithm and efficiency for large inputs
  - Bad: Ignores "four times faster"
- Systems equivalence: Account for constant overheads, performance tune
  - Good: Faster means different and better
  - Bad: Beware overtuning on "wrong" (e.g., small) inputs; definition does not let you "swap in a different algorithm"

Claim: Computer scientists implicitly (?) use all three every (?) day