CSE341: Programming Languages
Lecture 25
Subtyping for Records and Functions

Dan Grossman
Fall 2011
Last major course topic: more types

• SML and Java have static type systems to prevent some errors
  – ML: No such thing as a "treated number as function" error
  – Java: No such thing as a "message missing" error
  – Etc.

• What should the type of an object be?
  – Theory:
    • What fields it has (and what types of things they hold)
    • What methods it has (and argument/result types)
      – With Ruby style getters/setters, no need to treat fields separately
  – Common practice:
    • Use the names of classes and interfaces instead
      – Has plusses and minuses; see next lecture
Being more flexible

• ML's type system would be much more painful (reject safe programs you want to write) without \textit{parametric polymorphism}
  – Also known as \textit{generics}
  – Example: A separate length function for \texttt{int list} and \texttt{string list}

• Java's type system would be much more painful (reject safe programs you want to write) without \textit{subtype polymorphism}
  – Also known as \textit{subtyping}
  – Example: Couldn't pass an instance of a subtype when expecting an instance of a supertype
  – (Yes, Java also has generics as a separate concept)
So which is better?

- Generics and subtyping are best for different things
  - And you can combine them in interesting ways
  - But that's for next lecture because…

- First we need to learn how subtyping works!
  - Classes, interfaces, objects, methods, etc. will get in the way at first (we'll get there)
  - So start with just subtyping for records with mutable fields
  - We will make up our own syntax
    - ML has records, but no subtyping or field-mutation
    - Racket and Ruby have no type system
    - Java uses class/interface names and rarely fits on a slide
Records (half like ML, half like Java)

Record expression (field names and contents):
\[
\{f_1=e_1, f_2=e_2, \ldots, f_n=e_n\}
\]
Evaluate $e_i$, make a record

Record field access:
\[
e.f
\]
Evaluate $e$ to record $v$ with an $f$ field, get contents

Record field update
\[
e_1.f = e_2
\]
Evaluate $e_1$ to a record $v_1$ and $e_2$ to a value $v_2$;
Change $v_1$'s $f$ field (which must exist) to $v_2$;
Return $v_2$
A Basic Type System

Record types: What fields a record has and type of contents

{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):

```plaintext
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

```java
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

\[
\{f_1:t_1, f_2:t_2, \ldots, f_n:t_n\}
\]

Then it can also have a type missing some of the fields

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

```lang-ml
fun distToOrigin (p:{x:real,y:real}) = ...
fun makePurple (p:{color:string}) = ...
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 don't 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 \( t_1 <: t_2 \) for \( t_1 \) is a subtype of \( t_2 \)

– One new typing rule that uses subtyping:
  
  If \( e \) has type \( t_1 \) and \( t_1 <: t_2 \),
  
  then \( e \) (also) has type \( t_2 \)

So now we just have to define \( t_1 <: t_2 \)
Subtyping is not a matter of opinion

- Misconception: If we are making a new language, we can have whatever typing and subtyping rules we want

- Well, not if you want to prevent what you claim to prevent
  - Here: No accessing record fields that don't exist

- Our typing rules were *sound* before we added subtyping
  - So we better keep it that way

- Principle of *substitutability*: If \( \texttt{t1} <: \texttt{t2} \), then any value of type \( \texttt{t1} \) must be able to be used in every way a \( \texttt{t2} \) can be
  - Here: It needs all the same fields
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 $t_1 <: t_2$ and $t_2 <: t_3$, then $t_1 <: t_3$

4. Reflexivity: Every type is a subtype of itself

(4) may seem unnecessary, but it composes well with other rules in a full language and "can't hurt"
But this still is not allowed

[Warning: I'm tricking you into doing a bad thing 😊]

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

Example: A circle has a center field holding another record

```kotlin
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:

```kotlin
{center:{x:real,y:real,z:real}, r:real} <: {center:{x:real,y:real}, r:real}
```
Don't have this subtyping – could we?

\[
\{\text{center:}\{x:\text{real},y:\text{real},z:\text{real}\}, \ r:\text{real}\} \\
\quad <: \\
\{\text{center:}\{x:\text{real},y:\text{real}\}, \ r:\text{real}\}
\]

• No way to get this yet: we can drop \text{center}, drop \text{r}, or permute order, but we can't "reach into a field type" to do subtyping

• So why not add another subtyping rule… "Depth" subtyping: If \text{ta} <: \text{tb}, then \{f_1:t_1, \ldots, f:\text{ta}, \ldots, f_n:t_n\} <: \{f_1:t_1, \ldots, f:\text{tb}, \ldots, f_n:t_n\}

• Depth subtyping (along with width on the field's type) allows our example to type-check
  – Unfortunately, it also allows some things it should not… 😞
Mutation strikes again

If $ta \leq tb$, then $\{f_1:t_1, \ldots, f:ta, \ldots, fn:tn\} \leq \{f_1:t_1, \ldots, f:tb, \ldots, fn:tn\}$

```kotlin
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!
  – So this is the Nth time in the course we have seen a 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 $t_1 <: t_2$, then $t_1[] <: t_2[]$
- So this code type-checks, surprisingly

```java
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; // !
}
```
Why did they do this?

• More flexible type system allows more programs but prevents fewer errors
  – Seemed especially important before Java/C# had generics

• Good news, despite this "inappropriate" depth subtyping
  – e.color will never fail due to there being no color field
  – Array reads e1[e2] always return a (subtype of) t if e1 is a t[]

• Bad news, to get the good news given "inappropriate" subtyping
  – e1[e2]=e3 can fail even if e1 has type t[] and e3 has type t
  – Array stores check the run-time class of e1's elements and do not allow storing a supertype
  – No type-system help to avoid such bugs / performance cost
So what happens

void m1(Point[] pt_arr) {
    pt_arr[0] = new Point(3,4); // can throw
}
String m2(int x) {
    ColorPoint[] cpt_arr = new ColorPoint[x];
    ...
    m1(cpt_arr); // "inappropriate" depth subtyping
    ColorPoint c = cpt_arr[0]; // fine, cpt_arr
    // will always hold (subtypes of) ColorPoints
    return c.color; // fine, a ColorPoint has a color
}

• Causes code in m1 to throw an ArrayStoreException
  – It is awkward at best to blame this code
  – Benefit is run-time checks occur only on array stores, not on
    field accesses like c.color
null

• Array stores probably the most surprising choice for flexibility over static checking

• But null is the most common one in practice
  – null is not an object; it has no fields or methods
  – But Java and C# let it have any object type (backwards, huh?!)  
  – So, in fact, we do not have the static guarantee that evaluating e in e.f or e.m(...) produces an object that has an f or m
  – The "or null" caveat leads to run-time checks and errors, as you have surely noticed

• Sometimes null is very convenient (like ML's option types)
  – But having "can't be null" types in the language would be nice
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 \( t_1 \rightarrow t_2 \), can you pass a \( t_3 \rightarrow t_4 \) instead?
  – Important for understanding methods
    • An object type is a lot like a record type where "method positions" are immutable and have function types
    • Flesh out this connection next lecture, using our understanding of function subtyping
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 in 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"
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 mean ta -> t <: tb -> t
The other way works!

```haskell
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 of 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})

• \texttt{flipXMakeGreen} has type
  \{x:real\} \rightarrow \{x:real,y:real,color:string\}

• Fine to pass a function of such a type as function of type
  \{x:real,y:real\} \rightarrow \{x:real,y:real\}

• If \(t_3 <: t_1\) and \(t_2 <: t_4\), then \(t_1\rightarrow t_2 <: t_3\rightarrow t_4\)
This time with enthusiasm

- If $t_3 <: t_1$ and $t_2 <: t_4$, then $t_1 -> t_2 <: t_3 -> t_4$
  - Function subtyping contravariant in argument(s) and covariant in results

- Also essential for understanding subtyping and methods in OOP

- The most unintuitive concept in this course
  - Smart people often forget and convince themselves that covariant arguments are okay
  - These smart people are always mistaken
  - At times, you or your boss or your friend may do this
  - Remember: A guy with a PhD in PL jumped out and down insisting that function/method subtyping is always contravariant in its argument -- covariant is unsound