STLC in one slide

Expressions: \[ e ::= x | \lambda x. \ e | e \ e | c \]

Values: \[ v ::= \lambda x. \ e | c \]

Types: \[ \tau ::= \text{int} | \tau \to \tau \]

Contexts: \[ \Gamma ::= . | \Gamma, x : \tau \]

\[
\begin{align*}
e_1 & \to e_1' \\
e_2 & \to e_2' \\
e_1 \ e_2 & \to e_1' \ e_2 \\
v \ e_2 & \to v \ e_2' \\
(\lambda x. \ e) \ v & \to e\{v/x\}
\end{align*}
\]

\[
\begin{align*}
\Gamma & \vdash c : \text{int} \\
\Gamma & \vdash x : \Gamma(x)
\end{align*}
\]

\[
\begin{align*}
\Gamma, x : \tau_1 & \vdash e : \tau_2 \\
\Gamma & \vdash e_1 : \tau_1 \to \tau_2 \\
\Gamma & \vdash e_2 : \tau_1 \\
\Gamma & \vdash (\lambda x. \ e) : \tau_1 \to \tau_2 \\
\Gamma & \vdash e_1 \ e_2 : \tau_2
\end{align*}
\]
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
Having laid the groundwork…

• So far:
  – Our language (STLC) is tiny
  – We used heavy-duty tools to define it

• Now:
  – Add lots of things quickly
  – Because our tools are all we need

• And each addition will have the same form…
A method to our madness

• The plan
  – Add syntax
  – Add new semantic rules (including substitution)
  – Add new typing rules

• If our addition extends the syntax of types, then
  – New values (of that type)
  – Ways to make the new values
    • (called introduction forms)
  – Ways to use the new values
    • (called elimination forms)
Let bindings (CBV)

\[ e ::= \ldots \mid \text{let } x = e_1 \text{ in } e_2 \]
(no new values or types)

\[ e_1 \rightarrow e_1' \]

\[ \text{let } x = e_1 \text{ in } e_2 \rightarrow \text{let } x = e_1' \text{ in } e_2 \]

\[ \text{let } x = v \text{ in } e_2 \rightarrow e_2\{v/x\} \]

\[ \Gamma \vdash e_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash e_2 : \tau_2 \]

\[ \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_2 \]
Let as sugar?

Let is actually so much like lambda, we could use two other different but equivalent semantics

2. \texttt{let x=e1 in e2} is sugar (a different concrete way to write the same abstract syntax) for \((\lambda x.e2)\ e1\)

3. Instead of rules on last slide, use just

\[
\texttt{let x = e1 in e2} \rightarrow (\lambda x.e2)\ e1
\]

Note: In Caml, let is \textit{not} sugar for application because let is type-checked differently (type variables)
Booleans

\[
e ::= \ldots | \text{tru} | \text{fls} | e \, ? \, e : e
\]

\[
v ::= \ldots | \text{tru} | \text{fls}
\]

\[
\tau ::= \ldots | \text{bool}
\]

\[
e_1 \rightarrow e_1'
\]

\[
e_1 \, ? \, e_2 : e_3 \rightarrow e_1' \, ? \, e_2 : e_3
\]

\[
\Gamma \vdash \text{tru:bool}
\]

\[
\text{tru} \, ? \, e_2 : e_3 \rightarrow e_2
\]

\[
\Gamma \vdash \text{fls:bool}
\]

\[
\text{fls} \, ? \, e_2 : e_3 \rightarrow e_3
\]

\[
\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau
\]

\[
\Gamma \vdash e_1 \, ? \, e_2 : e_3 : \tau
\]
Caml? Large-step?

• In homework 3, you add conditionals, pairs, etc. to our environment-based large-step interpreter
• Compared to last slide
  – Different meta-language (cases rearranged)
  – Large-step instead of small
  – If tests an integer for 0 (like C)
• Large-step booleans with inference rules

\[
\begin{align*}
\text{tru} & \Downarrow \text{tru} \\
\text{fls} & \Downarrow \text{fls} \\
\text{e1} & \Downarrow \text{tru} \quad \text{e2} \Downarrow \text{v} \\
\text{e1} \text{ ? e2 : e3} & \Downarrow \text{v} \\
\end{align*}
\]
Pairs (CBV, left-to-right)

\[
\begin{align*}
  e & ::= \ldots | (e, e) | e.1 | e.2 \\
  v & ::= \ldots | (v, v) \\
  \tau & ::= \ldots | \tau^* \tau \\
  e_1 & \rightarrow e_1' \\
  e_2 & \rightarrow e_2' \\
  e & \rightarrow e' \\
  (e_1, e_2) & \rightarrow (e_1', e_2) \\
  (v, e_2) & \rightarrow (v, e_2') \\
  (v_1, v_2).1 & \rightarrow v_1 \\
  (v_1, v_2).2 & \rightarrow v_2 \\
  \Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 & \quad \Gamma \vdash e : \tau_1^* \tau_2 & \quad \Gamma \vdash e : \tau_1^* \tau_2 \\
  \Gamma \vdash (e_1, e_2) : \tau_1^* \tau_2 & \quad \Gamma \vdash e.1 : \tau_1 & \quad \Gamma \vdash e.2 : \tau_2
\end{align*}
\]
Toward Sums

• Next addition: sums (much like ML datatypes)

• Informal review of ML datatype basics

\[
\text{type } t = A \text{ of } t_1 \mid B \text{ of } t_2 \mid C \text{ of } t_3
\]

– Introduction forms: constructor-applied-to-exp
– Elimination forms: \text{match } e_1 \text{ with } \text{pat} - \rightarrow \text{exp} \ldots
– Typing: If e has type t_1, then A e has type t \ldots
Unlike ML, part 1

- Only patterns of form \( A \times (\text{rest is sugar}) \)
- Skip type parameters
- Avoid names (a bit simpler in theory)
- Add recursive types separately later

What we do will be simpler

- Allow fancy pattern matching
- Allow type parameters
- Introduce a new name for a type
- Allow recursive types
- ML datatypes do a lot at once
Unlike ML, part 2

- What we add will also be *different*
  - Only two constructors A and B
  - All sum types use these constructors
  - So A e can have any sum type allowed by e’s type
  - No need to declare sum types in advance
  - Like functions, will “guess types” in our rules

- This should still help explain what datatypes are

- After formalism, compare to C unions and OOP
The math (with type rules to come)

\[
\begin{align*}
e & ::= \ldots \mid A \, e \mid B \, e \mid \text{match } e \text{ with } A \, x \to e \mid B \, x \to e \\
v & ::= \ldots \mid A \, v \mid B \, v \\
\tau & ::= \ldots \mid \tau + \tau
\end{align*}
\]

\[
\begin{align*}
e & \to e' & e & \to e' & e1 & \to e1' \\
\hline
A \, e & \to A \, e' & B \, e & \to B \, e' & \text{match } e1 \text{ with } A \, x \to e2 \mid B \, y \to e3 & \to \text{match } e1' \text{ with } A \, x \to e2 \mid B \, y \to e3 \\
\hline
\text{match } A \, v \text{ with } A \, x \to e2 \mid B \, y \to e3 & \to e2\{v/x\} \\
\hline
\text{match } B \, v \text{ with } A \, x \to e2 \mid B \, y \to e3 & \to e3\{y/x\}
\end{align*}
\]
Low-level view

You can think of datatype values as “pairs”
- First component: A or B (or 0 or 1 if you prefer)
- Second component: “the data”
- e2 or e3 of match evaluated with “the data” in place of the variable
- This is all like Caml as in lecture 1
- Example values of type int + (int -> int):

\[
\begin{array}{c|c}
0 & 17 \\
\end{array}
\quad
\begin{array}{c|c}
1 & \lambda x. \lambda y. ["y",6] \\
\ \ \ | \ \ \\
\ \ \ \ \ \ x+y \\
\end{array}
\]
Typing rules

- Key idea for datatype exp: “other can be anything”
- Key idea for matches: “branches need same type”
  - Just like conditionals

\[
\begin{align*}
\Gamma & \vdash e : \tau_1 \\
\hline
\Gamma & \vdash A \ e : \tau_1 + \tau_2 \\
\Gamma & \vdash B \ e : \tau_1 + \tau_2 \\
\Gamma & \vdash e_1 : \tau_1 + \tau_2 \\
\Gamma, x : \tau_1 & \vdash e_2 : \tau \\
\Gamma, x : \tau_2 & \vdash e_3 : \tau \\
\hline
\Gamma & \vdash \text{match } e_1 \text{ with } A \ x \rightarrow e_2 \mid B \ y \rightarrow e_3 : \tau
\end{align*}
\]
Compare to pairs, part 1

- “pairs and sums” is a big idea
  - Languages should have both (in some form)
  - Somehow pairs come across as simpler, but they’re really “dual” (see Curry-Howard soon)
- Introduction forms:
  - pairs: “need both”, sums: “need one”

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 & \quad \Gamma \vdash e_2 : \tau_2 \\
\hline
\Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e : \tau_1 & \\
\hline
\Gamma \vdash e : \tau_2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash (e_1, e_2) : \tau_1 \ast \tau_2 & \\
\hline
\Gamma \vdash A e : \tau_1 + \tau_2 \\
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash B e : \tau_1 + \tau_2 & \\
\end{align*}
\]
Compare to pairs, part 2

- Elimination forms
  - pairs: “get either”, sums: “be prepared for either”

\[
\begin{align*}
\Gamma &\vdash e : \tau_1 \times \tau_2 & \Gamma &\vdash e : \tau_1 \times \tau_2 \\
\Gamma &\vdash e.1 : \tau_1 & \Gamma &\vdash e.2 : \tau_2 \\
\Gamma &\vdash e1 : \tau_1 + \tau_2 & \Gamma, x : \tau_1 &\vdash e2 : \tau & \Gamma, x : \tau_2&\vdash e3 : \tau \\
\end{align*}
\]

\[\Gamma \vdash \text{match e1 with A x -> e2 | B y -> e3 : } \tau\]
Living with just pairs

• If stubborn you can cram sums into pairs (don’t!)
  – Round-peg, square-hole
  – Less efficient (dummy values)
  • Flattened pairs don’t change that
  – More error-prone (may use dummy values)
  – Example: \( \texttt{int + (int -> int)} \) becomes
    \( \texttt{int * (int * (int -> int))} \)
Sums in other guises

```c
typedef struct { enum { A, B, C } tag; union { t1 a; t2 b; t3 c; } data; } t;
... switch (e->tag) { case A: t1 x=e->data.a; ...
```

- No static checking that tag is obeyed
- As fat as the fattest variant (avoidable with casts)
- Mutation costs us again!
- Shameless plug: Cyclone has ML-style datatypes
Sums in other guises

\[
\text{type } t = A \mid B \mid C \text{ of } t1 | t2 | t3
\]

\[
\text{match } e \text{ with } A x \rightarrow ...
\]

Meets Java:

\[
\begin{align*}
\text{abstract class } t & \{ \text{abstract Object } m(); \} \\
\text{class } A \text{ extends } t & \{ t1 x; \text{ Object } m(); \{ \ldots \} \} \\
\text{class } B \text{ extends } t & \{ t2 x; \text{ Object } m(); \{ \ldots \} \} \\
\text{class } C \text{ extends } t & \{ t3 x; \text{ Object } m(); \{ \ldots \} \}
\end{align*}
\]

\[
\ldots e.m() \ldots
\]

- A new method for each match expression
- Supports orthogonal forms of extensibility
  (will come back to this)
Where are we

• Have added let, bools, pairs, sums
• Could have done string, floats, records, …
• Amazing fact:
  – Even with everything we have added so far, every program terminates!
  – i.e., if \( \vdash e : \tau \) then there exists a value \( v \) such that \( e \rightarrow^* v \)
  – Corollary: Our encoding of fix won’t type-check
• To regain Turing-completeness, need explicit support for recursion
Recursion

- Could add “fix e” (ask me if you’re curious), but most people find “letrec f x e” more intuitive

\[
e ::= \ldots | \text{letrec } f \ x \ . \ e
\]

\[
v ::= \ldots | \text{letrec } f \ x \ . \ e
\]

(no new types)

“Substitute argument like lambda & whole function for f”

\[
\text{(letrec } f \ x \ . \ e) \ v \rightarrow (e\{v/x\})\{\text{(letrec } f \ x \ . \ e) / f\}
\]

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

\[
\Gamma \vdash \text{letrec } f \ x \ . \ e : \tau_1 \rightarrow \tau_2
\]
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 (∃), Recursive types
• Type inference
Static vs. dynamic typing

• First decide something is an error
  – Examples: 3 + “hi”, function-call arity
  – Examples: divide-by-zero, null-pointer dereference
• Then decide when to prevent the error
  – Example: At compile-time (static)
  – Example: At run-time (dynamic)
• “Static vs. dynamic” can be discussed rationally!
  – Most languages have some of both
  – There are trade-offs based on facts
Eagerness

I prefer to acknowledge a continuum rather than “static vs. dynamic” (2 most common points)
Example: divide-by-zero and code $x/0$
• Compile-time: reject if reachable code
  – e.g., dead branch
• Link-time: reject if reachable code
  – e.g., unused function
• Run-time: reject if code executed
  – e.g., maybe some branch never taken
• Later: reject if result is used to index an array?
  – cf. floating-point nan!
Exploring some arguments

1. “Dynamic/static typing” is more convenient
   Dynamic avoids “dinky little sum-types”
   
   ```
   let f x = if x>0 then 2*x else false
   vs.
   type t = A of int | B of bool
   let f x = if x>0 then A(2*x) else B false
   ```

   Static avoid “dinky little assertions”
   
   ```
   let f x = if int? x then … else raise …
   vs.
   let f (x:int) = if x>0 …
   ```
Exploring some arguments

2. Static typing does/doesn’t prevent useful programs

Overly restrictive type systems certainly can (no polymorphism, the Pascal array debacle)

Sum types give you as much flexibility as you want:

```haskell
type anything =
    Int of int
    | Bool of bool
    | Fun of anything -> anything
    | Pair of anything * anything
    | ...
```

Viewed this way, dynamic typing is static typing with one type and implicit tag addition/checking/removal
Exploring some arguments

3. Static/dynamic typing better for code evolution

If you change the type of something…

- Dynamic:
  - program still compiles
  - can incrementally evolve other code for the change?

- Static:
  - type-checker guides you to what must change
  - argument against wildcard patterns
Exploring some arguments

4. Sum types should/shouldn’t be *extensible*
   - New variants in other modules or at run-time

   • Dynamic:
     - Necessary for abstraction (branding)
     - Necessary for an evolving world (e.g., service discovery)
     - Even ML has one extensible type: exn

   • Static:
     - Can establish exhaustiveness at compile-time

   • Languages should have both? Which to use?
Exploring some arguments

5. Types make code reuse easier/harder

• Dynamic: Can write libraries that “crawl over every sort of data” (reflection trivial)

• Static: Whole point of segregating values by type is to avoid bugs from misuse

• In practice: Whether to encode with an existing type and use libraries (e.g., lists) or make a new type is a key design trade-off
Exploring some arguments

6. Types make programs slower/faster

• Dynamic:
  – Faster because don’t have to code around the type system
  – Optimizer can remove unnecessary tag tests

• Static
  – Faster because programmer controls tag tests
| Our plan |
|----------|--------------------------------|
| * Simply-typed Lambda-Calculus |
| * Safety = (preservation + progress) |
| * Extensions (pairs, datatypes, recursion, etc.) |
| * Digression: static vs. dynamic typing |
| * Type Variables: Generics (∧), Abstract types (∃), Recursive types |
| * Subtyping |
| * Type inference |
Curry-Howard Isomorphism

• What we did
  – Define a PL
  – Define a type system to filter out programs

• What logicians do
  – Define a logic, e.g.,
    \[ f ::= p \mid f \text{ or } f \mid f \text{ and } f \mid f \to f \]
  – Define a proof system

• But we did that too!
  – Types are formulas (propositions)
  – Programs are proofs
A funny STLC

• A strange language (no constants or fix, infinite number of “base types”)
• All our evaluation and typing rules from before

Expressions:  
\[ e ::= x \mid \lambda x.\ e \mid e\ e \mid (e, e) \mid e.1 \mid e.2 \]
\[ \mid A\ e \mid B\ e \mid \text{match } e \text{ with } A\ x \to e \mid B\ x \to e \]

Types:  
\[ \tau ::= p1 \mid p2 \mid \ldots \mid \tau \to \tau \mid \tau \cdot \tau \mid \tau + \tau \]

Now: Each typing rule is a proof rule from propositional logic (→ is implies, * is and, + is or)
(\( \Gamma \) is what we assume we have proofs for)
I’m not kidding

\[
\begin{align*}
\Gamma & \vdash x : \Gamma(x) \\
\Gamma, x : \tau_1 & \vdash e : \tau_2 \\
\Gamma & \vdash (\lambda x. e) : \tau_1 \rightarrow \tau_2 \\
\Gamma & \vdash e_1 : \tau_1 \quad \Gamma & \vdash e_2 : \tau_1 \\
\Gamma & \vdash e_1 : \tau_1 \quad \Gamma & \vdash e_2 : \tau_2 \\
\Gamma & \vdash (e_1, e_2) : \tau_1 \ast \tau_2 \\
\Gamma & \vdash e.1 : \tau_1 \quad \Gamma & \vdash e.2 : \tau_2
\end{align*}
\]
An exact isomorphism

• Our type system only proves true things!
  – An e such that \( \Gamma \vdash e : \tau \) is a proof of \( \tau \)

• Our type system can prove all true things except anything that implies “p or not p” (i.e., \( p_1 + (p_1 \rightarrow p_2) \))

• This is called constructive propositional logic
  – Programs have to “know how the world is”

• It’s not just this type system: For every constructive logic there’s a type system and vice-versa
What about fix

• letrec lets you prove anything
  – (that’s bad – an “inconsistent logic”)

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

\[ \Gamma \vdash \text{letrec } f \, x . \, e : \tau_1 \rightarrow \tau_2 \]

• Only terminating programs are proofs!
Why care?

• It’s just fascinating

• Now guides work on types and logic (brings fields closer together)

• Thinking the other way helps you know what’s possible (in ML a function of type `\texttt{\texttt{a} -> \texttt{b}}` does not return normally)

• Shows lambda-calculus is no more or less “made up” than logical proof systems
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 (∃), Recursive types
- 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 go beyond that
  – Ad hoc: $e1+e2$ in C less than Java less than C++
  – Parametric: “generics” $'a->'a$ can also have type $\text{int->int}$ or $(b->b)->(b->b)$
  – Subtype: If f takes an Object, can call f with a $C \leq \text{Object}$

• Try to avoid the ambiguous word polymorphism
• Will do subtyping first with records not objects
Records w/o polymorphism

Like pairs, but fields named and any number of them:
Field names: \( l \) (distinct from variables)

Exps: \[ e ::= \ldots | \{ l = e, \ldots, l = e \} | e . l \]

Types: \[ \tau ::= \ldots | \{ l = \tau, \ldots, l = \tau \} \]

\[ e \rightarrow e' \]

\[ \{ l_1 = v_1, \ldots, l_i = v_i, l_j = e, \ldots, l_n = e_n \} \rightarrow \{ l_1 = v_1, l_i = v_i, l_j = e', \ldots, l_n = e_n \} \]

\[ \{ l_1 = v_1, \ldots, l_i = v_i, \ldots, l_n = v_n \}. \ l_i \rightarrow v_i \]

\[ \Gamma \vdash e : \{ l_1 = \tau_1, \ldots, l_n = \tau_n \} \]

\[ \Gamma \vdash e . l_i : \tau_i \]

\[ \Gamma \vdash \{ l_1 = e_1, \ldots, l_n = e_n \} : \{ l_1 = \tau_1, \ldots, l_n = \tau_n \} \]
Width

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

\[
\begin{align*}
\text{let } f &= \lambda x. \text{\textit{x.11 + x.12}} \text{ in } (* f : \{11=\text{int}, \text{12=\text{int}}\} -\rightarrow \text{int} *) \\
&\quad (f \{11=3, \text{12=4}\}) + (f \{11=7, \text{12=8}, \text{13=9}\})
\end{align*}
\]

- \( f \) has to have one type, but \textit{wider} arguments okay
- New judgment:  \( \tau_1, \tau_2 \)
- A rule for this judgment (more later):

\[
\begin{align*}
\text{\{11=\tau_1, ..., ln=\tau_n\}} & \quad \text{\{11=\tau_1, ..., ln=\tau_n, l=\tau\}}
\end{align*}
\]

- (Allows 1 new field, but we’ll be able to use the rule multiple times)
Using it

• Haven’t done anything until we add the all-purpose *subsumption* rule for our *type-checking judgment*:

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

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

• Width + subsumption lets us typecheck last example

• To add multiple fields, use a *transitivity rule* for our *subtyping judgment*

\[ \tau_1 \quad \tau_2 \quad \tau_2 \quad \tau_3 \]

\[ \tau_1 \quad \tau_3 \]
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

\{l_1 = \tau_1, \ldots, l_i = \tau_i, l_j = \tau_j, l_n = \tau_n\}
\{l_1 = \tau_1, \ldots, l_j = \tau_j, l_i = \tau_i, l_n = \tau_n\}

• Note in passing: Efficient algorithms to decide if \(\tau_1 \quad \tau_2\) are not always simple or existent
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 e. \( \mathbf{1} \) 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
Toward depth

Recall we added width to type-check this code:

\[
\text{let } f = \lambda x. x.11 + x.12 \text{ in } \text{(* } f : \{11=\text{int}, 12=\text{int}\} -\rightarrow \text{ int } \text{*)}
\]
(\text{f \{11=3, 12=4\}}) + (\text{f \{11=7, 12=8, 13=9\}})

But we still can’t type-check this code:
\[
\text{let } f = \lambda x. x.1.11 + x.1.12 \text{ in }
\]
(\text{f \{1 = \{11=3, 12=4\}\}}) + (\text{f \{1 =\{11=7, 12=8, 13=9\}\} \}})

Want subtyping “deeper” in record types…
Depth

• This rule suffices

\[
\tau_1 \quad \tau
\]

\[
\{l_1 = \tau_1, \ldots, l_i = \tau_i, \ldots, l_n = \tau_n\}
\]

• 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!
  – Will come back to why immutability is key (hw?)
Toward function subtyping

- So far allow some record types where others expected
- What about allowing some function types where others expected
- For example,
  \[ \text{int} \to \{ l_1 = \text{int}, l_2 = \text{int} \} \quad \text{int} \to \{ l_1 = \text{int} \} \]
- But what’s the general principle?
  
  \[ \tau_1 \to \tau_2 \quad \tau_3 \to \tau_4 \]
Function subtyping

\( \tau_3 \quad \tau_1 \quad \tau_2 \quad \tau_4 \)

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

Also want: \[ \tau \quad \tau \]

- Supertype can impose more restrictions on arguments and reveal less about results
- Jargon: Contravariant in argument, covariant in result
- Example:
  \[ \{ l_1 = \text{int}, l_2 = \text{int} \} \rightarrow \{ l_1 = \text{int}, l_2 = \text{int} \} \]
  \[ \{ l_1 = \text{int}, l_2 = \text{int}, l_3 = \text{int} \} \rightarrow \{ l_1 = \text{int} \} \]
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!!!
  – If I jump up and down will you remember “slide 51, lecture 6”?
Where are we

- So far: Added subsumption and subtyping rules

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

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

- Immutable records: width, permutation, depth
  - Depth = covariant fields
- Functions: contravariant arg, 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 {ll= int} “autoboxing”
3. ℝ 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` `{11= int}` and `==` is bit-equality for all types
It’s a mess

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#)

```cpp
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:

\[
\text{if}\_\text{hastype}(\tau, e_1) \text{ then } x \rightarrow e_2 \text{ else } e_3
\]

“Roughly, if at run-time \(e_1\) has type \(\tau\) (or a subtype), then bind it to \(x\) and evaluate \(e_2\). Else evaluate \(e_3\).”

• Just to show the issue is orthogonal to exceptions

• In Java you use \text{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 \(f : \{l1=\text{int}\} \rightarrow \text{int}\) 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 (∀), Abstract types (∃), Recursive types
• Type inference