CSEP505: Programming Languages

Lecture 6: Types, Subtyping

Dan Grossman

Spring 2006

STLC in one slide

Expressions: $e ::= x \mid \lambda x.\ e \mid e\ e \mid c$

Values: $v ::= \lambda x.\ e \mid c$

Types: $\tau ::= \mathtt{int} \mid \tau \rightarrow \tau$

Contexts: $\Gamma ::= . \mid \Gamma, x : \tau$

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

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

$$\frac{\Gamma, x{:}\ \tau1 \vdash e : \tau2}{\Gamma \vdash (\lambda x.e) : \tau1 \rightarrow \tau2} \qquad \frac{\Gamma \vdash e1 : \tau1 \rightarrow \tau2 \quad \Gamma \vdash e2 : \tau1}{\Gamma \vdash e1\ e2 : \tau2}$$

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 ::= … | let x = e1 in e2`

(no new values or types)

$$\frac{\texttt{e1} \rightarrow \texttt{e1'}}{\texttt{let x = e1 in e2} \rightarrow \texttt{let x = e1' in e2}}$$

$$\frac{}{\texttt{let x = v in e2} \rightarrow \texttt{e2}\{\texttt{v/x}\}}$$

$$\frac{\Gamma \vdash \texttt{e1}:\tau 1 \qquad \Gamma,\texttt{x}:\tau 1 \vdash \texttt{e2}:\tau 2}{\Gamma \vdash \texttt{let x = e1 in e2} : \tau 2}$$

Let as sugar?

Let is actually so much like lambda, we could use 2 other different but equivalent semantics

2. let x=e1 in e2 is sugar (a different concrete way to write the same abstract syntax) for (λx.e2) e1

3. Instead of rules on last slide, use just

$$\frac{}{\texttt{let x = e1 in e2} \rightarrow \texttt{(}\lambda\texttt{x.e2) e1}}$$

Note: In Caml, let is *not* sugar for application because let is type-checked differently (type variables)

Booleans

e ::= … | tru | fls | e ? e : e
v ::= … | tru | fls
τ ::= … | bool

$$\frac{\text{e1} \rightarrow \text{e1'}}{\text{e1 ? e2 : e3} \rightarrow \text{e1' ? e2 : e3}}$$

$$\frac{}{\text{tru ? e2 : e3} \rightarrow \text{e2}}$$

$$\frac{}{\text{fls ? e2 : e3} \rightarrow \text{e3}}$$

$$\frac{}{\Gamma \vdash \text{tru} : \text{bool}}$$

$$\frac{}{\Gamma \vdash \text{fls} : \text{bool}}$$

$$\frac{\Gamma \vdash \texttt{e1} : \texttt{bool} \quad \Gamma \vdash \texttt{e2} : \tau \quad \Gamma \vdash \texttt{e3} : \tau}{\Gamma \vdash \text{e1 ? e2 : e3} : \tau}$$

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

$$\frac{}{\text{tru} \Downarrow \text{tru}} \qquad \frac{}{\text{fls} \Downarrow \text{fls}}$$

$$\frac{e1 \Downarrow \text{tru} \quad e2 \Downarrow v}{e1 \; ? \; e2 : e3 \Downarrow v} \qquad \frac{e1 \Downarrow \text{fls} \quad e3 \Downarrow v}{e1 \; ? \; e2 : e3 \Downarrow v}$$

Pairs (CBV, left-to-right)

$$
\begin{aligned}
e &::= \ldots \mid (e,e) \mid e.1 \mid e.2 \\
v &::= \ldots \mid (v,v) \\
\tau &::= \ldots \mid \tau * \tau
\end{aligned}
$$

$$\frac{e1 \to e1'}{(e1,e2) \to (e1',e2)} \qquad \frac{e2 \to e2'}{(v,e2) \to (v,e2')} \qquad \frac{e \to e'}{e.1 \to e'.1} \qquad \frac{e \to e'}{e.2 \to e'.2}$$

$$\frac{}{(v1,v2).1 \to v1} \qquad \frac{}{(v1,v2).2 \to v2}$$

$$\frac{\Gamma \vdash e1 : \tau1 \quad \Gamma \vdash e2 : \tau2}{\Gamma \vdash (e1,e2) : \tau1 * \tau2} \qquad \frac{\Gamma \vdash e : \tau1 * \tau2}{\Gamma \vdash e.1 : \tau1} \qquad \frac{\Gamma \vdash e : \tau1 * \tau2}{\Gamma \vdash e.2 : \tau2}$$

Toward Sums

- Next addition: sums (much like ML datatypes)
- Informal review of ML datatype basics

```
type t = A of t1 | B of t2 | C of t3
```

  - Introduction forms: constructor-applied-to-exp
  - Elimination forms: match e1 with pat -> exp …
  - Typing: If e has type t1, then A e has type t …

Unlike ML, part 1

- ML datatypes do a lot at once
  - Allow recursive types
  - Introduce a new *name* for a type
  - Allow type parameters
  - Allow fancy pattern matching
- What we do will be *simpler*
  - Add recursive types separately later
  - Avoid names (a bit simpler in theory)
  - Skip type parameters
  - Only patterns of form A x (rest is sugar)

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)

e ::= … | A e | B e | match e with A x -> e | B x -> e

v ::= … | A v | B v

τ ::= … | τ+τ

$$\frac{e \rightarrow e'}{A\ e \rightarrow A\ e'} \qquad \frac{e \rightarrow e'}{B\ e \rightarrow B\ e'} \qquad \frac{e1 \rightarrow e1'}{\begin{array}{l}\text{match } e1 \text{ with A } x\text{->}e2 \mid \text{B } y \text{->} e3\\ \rightarrow \text{ match } e1' \text{ with A } x\text{->}e2 \mid \text{B } y \text{->} e3\end{array}}$$

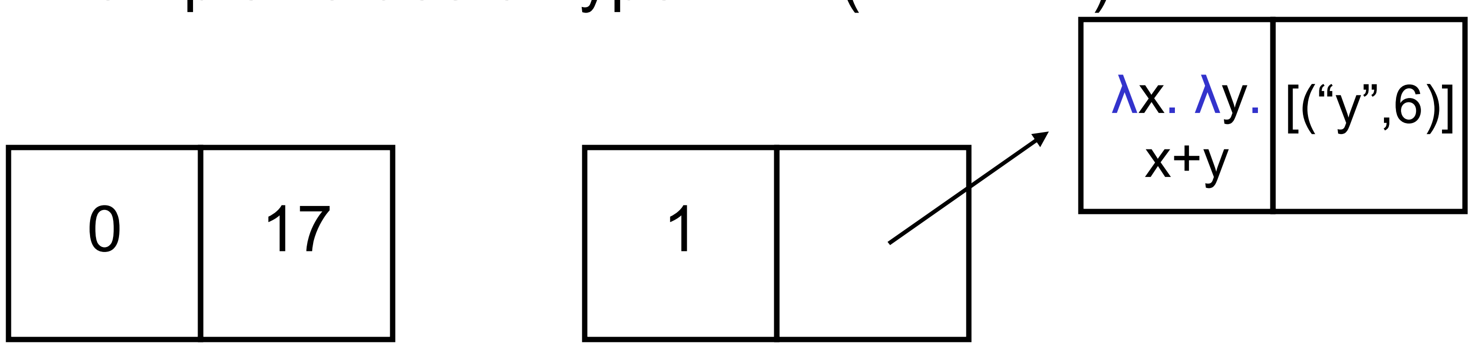
$$\frac{}{\text{match A } v \text{ with A } x\text{->}e2 \mid \text{B } y \text{->} e3 \ \rightarrow\ e2\{v/x\}}$$

$$\frac{}{\text{match B } v \text{ with A } x\text{->}e2 \mid \text{B } y \text{->} e3 \ \rightarrow\ e3\{y/x\}}$$

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):



Typing rules

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

$$\frac{\Gamma \vdash e : \tau 1}{\Gamma \vdash \mathrm{A}\ e : \tau 1 + \tau 2} \qquad \frac{\Gamma \vdash e : \tau 2}{\Gamma \vdash \mathrm{B}\ e : \tau 1 + \tau 2}$$

$$\frac{\Gamma \vdash e1 : \tau 1 + \tau 2 \qquad \Gamma, x{:}\tau 1 \vdash e2 : \tau \qquad \Gamma, x{:}\tau 2 \vdash e3 : \tau}{\Gamma \vdash \texttt{match}\ e1\ \texttt{with}\ \mathrm{A}\ x \texttt{->} e2 \mid \mathrm{B}\ y \texttt{->} e3 : \tau}$$

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"

$$\frac{\Gamma \vdash e1 : \tau1 \quad \Gamma \vdash e2 : \tau2}{\Gamma \vdash (e1,e2) : \tau1 * \tau2} \qquad \frac{\Gamma \vdash e : \tau1}{\Gamma \vdash A\ e : \tau1 + \tau2} \qquad \frac{\Gamma \vdash e : \tau2}{\Gamma \vdash B\ e : \tau1 + \tau2}$$

Compare to pairs, part 2

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

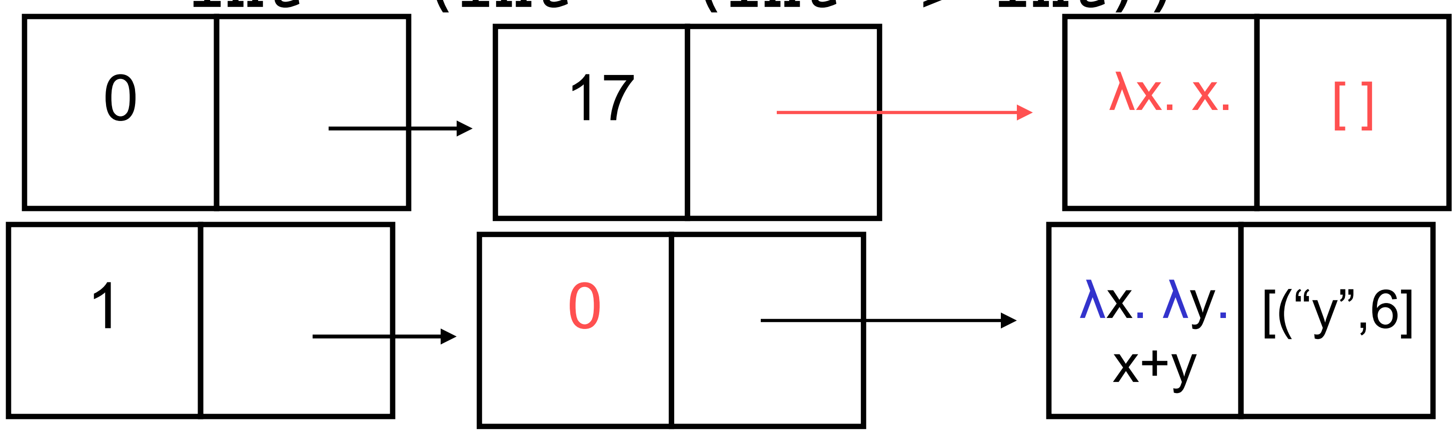
$$\frac{\Gamma \vdash \mathtt{e} : \tau 1 * \tau 2}{\Gamma \vdash \mathtt{e.1} : \tau 1} \qquad \frac{\Gamma \vdash \mathtt{e} : \tau 1 * \tau 2}{\Gamma \vdash \mathtt{e.2} : \tau 2}$$

$$\frac{\Gamma \vdash e1 : \tau 1 + \tau 2 \qquad \Gamma, x : \tau 1 \vdash e2 : \tau \qquad \Gamma, x : \tau 2 \vdash e3 : \tau}{\Gamma \vdash \mathtt{match}\ e1\ \mathtt{with}\ A\ x \texttt{->} e2 \mid B\ y \texttt{->} e3 : \tau}$$

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: **`int + (int -> int)`** becomes

    `int * (int * (int -> int))`



Sums in other guises

```
type t = A of t1 | B of t2 | C of t3
match e with A x -> …
```

Meets C:

```
 struct t {
  enum  {A, B, C}            tag;
  union {t1 a; t2 b; t3 c;} data;
 };
… 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

```
type t = A of t1 | B of t2 | C of t3
match e with A x -> …
```

Meets Java:

```
abstract class t {abstract Object m();}
class A extends t { t1 x; Object m(){…}}
class B extends t { t2 x; Object m(){…}}
class C extends t { t3 x; Object m(){…}}
… e.m() …
```

- 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 ::= … | letrec f x . e

v ::= … | letrec f x . e

(no new types)

"Substitute argument like lambda & whole function for f"

$$\frac{}{(\text{letrec } f\ x\ .\ e)\ v \rightarrow (e\{v/x\})\{(\text{letrec } f\ x\ .\ e)\ /\ f\}}$$

$$\frac{\Gamma, f : \tau 1 \rightarrow \tau 2, x : \tau 1 \vdash e : \tau 2}{\Gamma \vdash \text{letrec } f\ x\ .\ e : \tau 1 \rightarrow \tau 2}$$

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 ($\forall$), Abstract types ($\exists$), 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:

```
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
- Digression: Curry-Howard Isomorphism
- Subtyping
- Type Variables:
  - Generics ($\forall$), Abstract types ($\exists$), Recursive types
- 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 | f or f | f and f | f -> 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 | λx. e | e e | (e,e) | e.1 | e.2
| A e | B e | match e with A x->e |B x->e

Types τ ::= p1 | p2 | … | τ → τ | τ*τ | τ+τ

Now: Each typing rule is a proof rule from propositional logic (→ is implies, * is and, + is or)

(Γ is what we assume we have proofs for)

I'm not kidding

$$\frac{}{\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x})}$$

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

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

...

An exact isomorphism

- Our type system only proves true things!
  - An e such that $. \vdash e : \tau$ is a proof of τ
- Our type system can prove all true things except anything that implies "p or not p" (i.e., p1 + (p1 → p2))
- 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")

$$\frac{\Gamma, f : \tau 1 \rightarrow \tau 2, x : \tau 1 \vdash e : \tau 2}{\Gamma \vdash \text{letrec } f\ x\ .\ e : \tau 1 \rightarrow \tau 2}$$

- 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 **`’a->’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 ($\forall$), Abstract types ($\exists$), Recursive 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 go beyond that
  - *Ad hoc*: e1+e2 in C less than Java less than C++
  - *Parametric*: "generics" **`'a->'a`** can also have type **`int->int`** or **`('b->'b)->('b->'b)`**
  - *Subtype*: If f takes an Object, can call f with a C≤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: $e ::= \ldots \mid \{l=e, \ldots, l=e\} \mid e.l$

Types: $\tau ::= \ldots \mid \{l=\tau, \ldots, l=\tau\}$

$$\frac{e \rightarrow e'}{\{l1=v1, \ldots, li=vi, lj=e, \ldots, ln=en\} \rightarrow \{l1=v1, li=vi, lj=e', \ldots, ln=en\}}$$

$$\frac{e \rightarrow e'}{e.l \rightarrow e'.l}$$

$$\frac{}{\{l1=v1,\ldots,li=vi,\ldots,ln=vn\}.\ li \rightarrow vi}$$

$$\frac{\Gamma \vdash e : \{l1=\tau1,\ldots,ln=\tau n\}}{\Gamma \vdash e.\ li : \tau i}$$

$$\frac{\Gamma \vdash e1 : \tau1 \quad \ldots \quad \Gamma \vdash en : \tau n \qquad \text{“labels distinct”}}{\Gamma \vdash \{l1=e1, \ldots, ln=en\} : \{l1=\tau1,\ldots,ln=\tau n\}}$$

Width

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

let f = λx. x.*l1* + x.*l2* in (* f : {*l1*=int, *l2*=int}-> int *)

(f {*l1*=3, *l2*=4}) + (f {*l1*=7, *l2*=8, *l3*=9})

- f has to have one type, but wider arguments okay
- New judgment: $\tau 1 \quad \tau 2$
- A rule for this judgment (more later):

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

- (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*:

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

- Width + subsumption lets us typecheck last example
- To add multiple fields, use a *transitivity rule* for our subtyping judgment

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

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

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

- 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. *`l`* 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:

let f = λx. x.*l1* + x.*l2* in (* f : {*l1*=int, *l2*=int}-> int *)

(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.*l*.*l1* + x.*l*.*l2* in

(f {*l* = {*l1*=3, *l2*=4} }) + (f {*l* ={*l1*=7, *l2*=8, *l3*=9} })

Want subtyping “deeper” in record types…

Depth

- This rule suffices

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

- 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!
  - 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,

$\text{int} \rightarrow \{l1 = \text{int}, l2 = \text{int}\} \qquad \text{int} \rightarrow \{l1 = \text{int}\}$

- But what's the general principle?

$$\frac{??????}{\tau 1 \rightarrow \tau 2 \qquad \tau 3 \rightarrow \tau 4}$$

Function subtyping

$$\frac{\tau 3 \quad \tau 1 \qquad \tau 2 \quad \tau 4}{\tau 1 \rightarrow \tau 2 \qquad \tau 3 \rightarrow \tau 4} \qquad \text{Also want: } \frac{}{\tau \quad \tau}$$

- Supertype can impose more restrictions on arguments and reveal less about results
- Jargon: Contravariant in argument, covariant in result
- Example:

$\{l1 = \text{int}, l2 = \text{int}\} \rightarrow \{l1 = \text{int}, l2 = \text{int}\}$

$\{l1 = \text{int}, l2 = \text{int}, l3 = \text{int}\} \rightarrow \{l1 = \text{int}\}$

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!!!
  - If I jump up and down will you remember "slide 51, lecture 6"?

Where are we

- So far: Added subsumption and subtyping rules

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

- *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 {*l1*= int} "autoboxing"
3. `τ` 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 τ string
   - Can "fix" by printing ints with trailing .0
2. 34==34 where int {l1= 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#)

```
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 tags; 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 ($\forall$), Abstract types ($\exists$), Recursive types
- Type inference