CSEP505: Programming Languages
Lecture 6: Types, Types, Types

Dan Grossman

Autumn 2016

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 ($\exists$)
- Type inference

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

$$\boxed{e \rightarrow e'}$$

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

$$\boxed{\Gamma \vdash e : \tau}$$

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

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

Rule-by-rule

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

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

- Constant rule: context irrelevant
- Variable rule: lookup (no instantiation if x not in Γ)
- Application rule: “yeah, that makes sense”
- Function rule the interesting one…

The function rule

$$\frac{\Gamma, x{:}\tau 1 \vdash e : \tau 2}{\Gamma \vdash (\lambda x.e) : \tau 1 \rightarrow \tau 2}$$

- Where did $\tau 1$ come from?
  - Our rule “inferred” or “guessed” it
  - To be syntax-directed, change $\lambda x.e$ to $\lambda x{:}\tau.e$ and use that τ
- If we think of Γ as a partial function, we need x not already in it (implicit systematic renaming [alpha-conversion] allows)
  - Or can think of it as shadowing

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

Is it “right”?

- Can define any type system we want
- What we defined is sound and incomplete
- Can prove incomplete with one example
  - Every variable has exactly one simple type
  - Example (doesn’t get stuck, doesn’t typecheck)

    (λx. (x (λy.y)) (x 3)) (λz.z)

Sound

- Statement of soundness theorem:
  If $. \vdash \mathbf{e} : \tau$ and $\mathbf{e} \rightarrow^* \mathbf{e2}$,
  then **e2** is a value or there exists an **e3** such that $\mathbf{e2} \rightarrow \mathbf{e3}$
- Proof is non-trivial
  - Must hold for all **e** and any number of steps
  - But easy given two helper theorems…

1. Progress: If $. \vdash \mathbf{e} : \tau$, then **e** is a value or there exists an **e2** such that $\mathbf{e} \rightarrow \mathbf{e2}$
2. Preservation: If $. \vdash \mathbf{e} : \tau$ and $\mathbf{e} \rightarrow \mathbf{e2}$, then $. \vdash \mathbf{e2} : \tau$

Let's prove it

Prove: If $.\vdash \mathtt{e}:\tau$ and $\mathtt{e} \rightarrow^* \mathtt{e2}$,

then e2 is a value or $\exists$ $\mathtt{e3}$ such that $\mathtt{e2}\rightarrow\mathtt{e3}$, assuming:

1. If $.\vdash \mathtt{e}:\tau$ then $\mathtt{e}$ is a value or $\exists$ $\mathtt{e2}$ such that $\mathtt{e}\rightarrow\mathtt{e2}$
2. If $.\vdash \mathtt{e}:\tau$ and $\mathtt{e}\rightarrow\mathtt{e2}$ then $.\vdash \mathtt{e2}:\tau$

Prove something stronger: Also show $.\vdash \mathtt{e2}:\tau$

Proof: By induction on n where $\mathtt{e}\rightarrow^*\mathtt{e2}$ in n steps

- Case n=0: immediate from progress ($\mathtt{e}=\mathtt{e2}$)
- Case n>0: then $\exists\mathtt{e3}$ such that…

What's the point

- Progress is what we care about
- But Preservation is the *invariant* that holds no matter how long we have been running
- (Progress and Preservation) implies Soundness

- This is a very general/powerful recipe for showing you "don't get to a bad place"
  - If invariant holds, then (a) you're in a good place (progress) and (b) anywhere you go is a good place (preservation)

- Details on next two slides less important…

Forget a couple things?

Progress: If $\cdot \vdash$ **e : τ** then **e** is a value or there exists an **e2** such that **e**→**e2**

Proof: Induction on height of derivation tree for $\cdot \vdash$ **e : τ**

Rough idea:

- Trivial unless **e** is an application
- For **e** = **e1 e2**,
  - If left or right not a value, induction
  - If both values, **e1** must be a lambda…

Forget a couple things?

Preservation: If $. \vdash \mathbf{e} : \tau$ and $\mathbf{e} \to \mathbf{e2}$ then $. \vdash \mathbf{e2} : \tau$

Also by induction on assumed typing derivation.

The trouble is when $e \longrightarrow e'$ involves substitution

- Requires another theorem

Substitution:

If $\Gamma, \mathbf{x}:\tau\mathbf{1} \vdash \mathbf{e} : \tau$ and $\Gamma \vdash \mathbf{e1} : \tau\mathbf{1}$,

then $\Gamma \vdash \mathbf{e\{e1/x\}} : \tau$

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
  - Add new typing rules
    - Such that we remain safe

- 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)

$$\texttt{e} ::= \ldots \mid \texttt{let x = e1 in e2}$$

(no new values or types)

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

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

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

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, just use

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

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

Booleans

$$e \;::=\; \ldots \mid \texttt{tru} \mid \texttt{fls} \mid e\;?\;e : e$$
$$v \;::=\; \ldots \mid \texttt{tru} \mid \texttt{fls}$$
$$\tau \;::=\; \ldots \mid \texttt{bool}$$

$$\frac{e1 \to e1'}{e1\;?\;e2 : e3 \to e1'\;?\;e2 : e3}$$

$$\frac{}{\texttt{tru}\;?\;e2 : e3 \to e2} \qquad \frac{}{\texttt{fls}\;?\;e2 : e3 \to e3}$$

$$\frac{}{\Gamma \vdash \texttt{tru} : \texttt{bool}} \qquad \frac{}{\Gamma \vdash \texttt{fls} : \texttt{bool}}$$

$$\frac{\Gamma \vdash e1 : \texttt{bool} \qquad \Gamma \vdash e2 : \tau \qquad \Gamma \vdash e3 : \tau}{\Gamma \vdash e1\;?\;e2 : e3 : \tau}$$

OCaml? 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
- Large-step booleans with inference rules:

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

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

Pairs (CBV, left-to-right)

```
e  ::= … | (e,e) | e.1 | e.2
v  ::= … | (v,v)
τ  ::= … | τ*τ
```

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

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

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

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 expression
  - 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*
  - Skip recursive types [an orthogonal addition]
  - Avoid names (a bit simpler in theory)
  - Skip type parameters
  - Only patterns of form **`A x`** and **`B 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 still helps 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

$$\frac{e \rightarrow e'}{\text{A } e \rightarrow \text{A } e'} \qquad \frac{e \rightarrow e'}{\text{B } e \rightarrow \text{B } e'}$$

$$\frac{e1 \rightarrow e1'}{\text{match } e1 \text{ with A } x\text{->}e2 \mid \text{B } y \text{ -> } e3 \rightarrow \text{match } e1' \text{ with A } x\text{->}e2 \mid \text{B } y \text{ -> } e3}$$

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

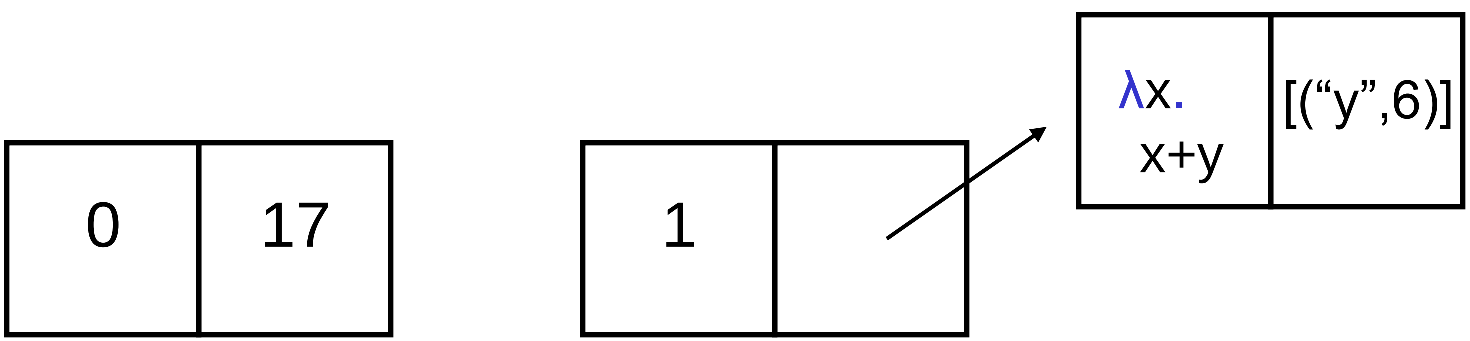
$$\frac{}{\text{match B } v \text{ with A } x\text{->}e2 \mid \text{B } y \text{ -> } e3 \rightarrow e3\{v/y\}}$$

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 OCaml as in Lecture 1

- Example values of type `int + (int -> int)`:



Typing rules

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

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

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

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"

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

Compare to pairs, part 2

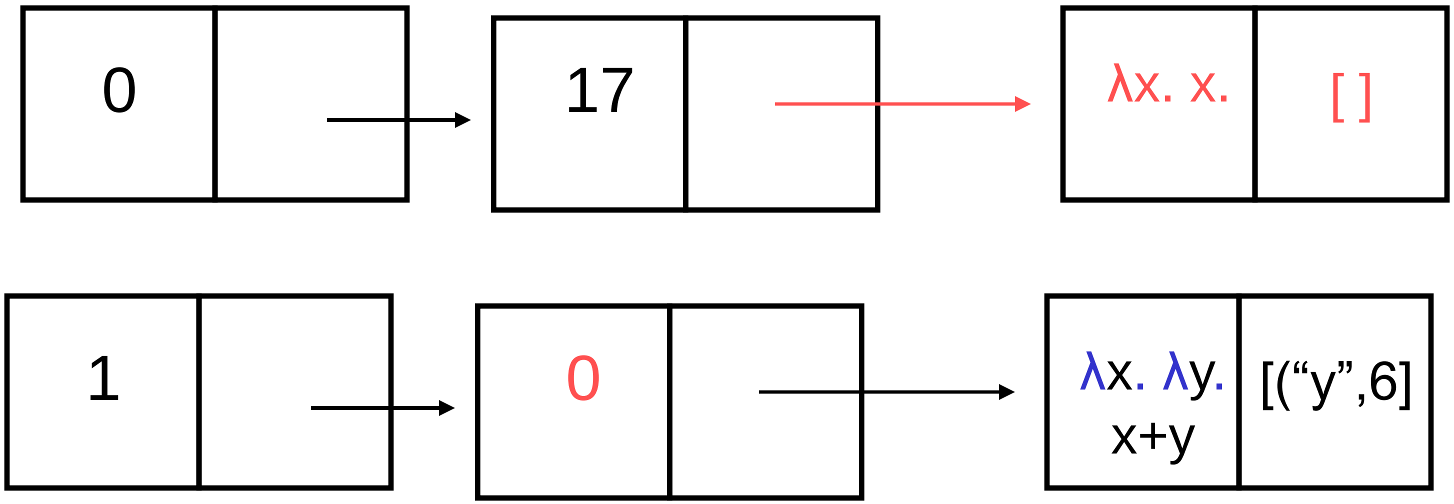
- Elimination forms
  - pairs: “get either”, sums: “be prepared for either”

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

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

Living with just pairs

- If stubborn you can cram sums into pairs (don't!)
  - Round-peg, square-hole
  - Less efficient (dummy values)
  - 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!

- Some "modern progress" in Rust, Swift, …?

Sums in other guises

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

Meets Java [C# similar]:

```
 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
  - New constructors vs. new operations over the dataype!

Where are we

- Have added let, bools, pairs, sums
- Could have added many other things
- Amazing fact:
  - Even with everything we have added so far, every program terminates!
  - i.e., if $. \vdash \mathbf{e} : \tau$ then there exists a value $\mathbf{v}$ such that

    $\mathbf{e} \rightarrow^{*} \mathbf{v}$
  - Corollary: Our encoding of recursion won't type-check
- To regain Turing-completeness, need explicit support for recursion

Recursion

- Could add "fix e", 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 ($\exists$)
- Type inference

Static vs. dynamic typing

- First decide something is an error
  - Examples: 3 + “hi”, function-call arity, redundant matches
  - Examples: divide-by-zero, null-pointer dereference, bounds
  - Soundness / completeness depends on what’s checked!
- 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

Basic benefits/limitations

Indisputable facts:

- Languages with static checks catch certain bugs without testing
  - Earlier in the software-development cycle

- Impossible to catch exactly the buggy programs at compile-time
  - Undecidability: even code reachability
  - Context: Impossible to know how code will be used/called
  - Application level: Algorithmic bugs remain
    - No idea what program you're trying to write

Eagerness

I prefer to acknowledge a continuum

- rather than "static vs. dynamic" (2 most common points)

Example: divide-by-zero and code **`3/0`**

- Keystroke time: Disallow it in the editor
- Compile-time: reject if code is reachable
  - maybe on a dead branch
- Link-time: reject if code is reachable
  - maybe function is never used
- Run-time: reject if code executed
  - maybe branch is never taken
- Later: reject only if result is used to index an array
  - cf. floating-point **`+inf.0`**!

Inherent Trade-off

“Catching a bug before it matters”

is in inherent tension with

“Don’t report a bug that might not matter”

- Corollary: Can always wish for a slightly better trade-off for a particular code-base at a particular point in time

Exploring some arguments

1. (a) “Dynamic typing is more convenient”
   - Avoids “dinky little sum types”

```
(* if OCaml were dynamically typed *)
let f x = if x>0 then 2*x else false
…
let ans = (f 19) + 4
```

versus

```
(* actual OCaml *)
type t = A of int | B of bool
let f x = if x>0 then A(2*x) else B false
…
let ans = match f 19 with A x -> x + 4
                        | _   -> raise Failure
```

Exploring some arguments

1. (b) "Static typing is more convenient"
   - Harder to write a library defensively that raises errors before it's too late or client gets a bizarre failure message

```
(* if OCaml were dynamically typed *)
let cube x = if int? x
             then x*x*x
             else raise Failure
```

versus

```
(* actual OCaml *)
let cube x = x*x*x
```

Exploring some arguments

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

Overly restrictive type systems certainly can (cf. Pascal arrays)

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

- Easy to compile dynamic typing into OCaml this way
- More painful by hand (constructors and matches *everywhere*)

Exploring some arguments

3. (a) Static catches bugs earlier
   - As soon as compiled
   - Whatever is checked need not be tested for
   - Programmers can “lean on the the type-checker”

Example: currying versus tupling:

```
(* does not type-check *)
let pow x y = if y=0
              then 1
              else x * pow (x,y-1)
```

Exploring some arguments

3. (b) But static often catches only "easy" bugs
   – So you still have to test
   – And any decent test-suite will catch the "easy" bugs too

Example: still wrong even after fixing currying vs. tupling

```
(* does not type-check and wrong algorithm *)
let pow x y = if y=0
              then 1
              else x + pow (x,y-1)
```

Exploring some arguments

4. (a) "Dynamic typing better for code evolution"

Imagine changing: `let cube x = x*x*x`

To:
```
type t = I of int | S of string
let cube x = match x with I i -> i*i*i
                        | S s -> s^s^s
```

– Static: Must change all existing callers

Dynamic: No change to existing callers…
```
let cube x = if int? x then x*x*x
                       else x^x^x
```

Exploring some arguments

4. (b) "Static typing better for code evolution"

Imagine changing the return type instead of the argument type:

```
 let cube x = if x > 0 then I (x*x*x)
                          else S "hi"
```

- Static: Type-checker gives you a full to-do list
  - cf. Adding a new constructor if you avoid wildcard patterns

- Dynamic: No change to existing callers; failures at runtime

```
let cube x = if x > 0 then x*x*x
                         else "hi"
```

Exploring some arguments

5. Types make code reuse easier/harder

- Dynamic:
  - Sound static typing always means some code could be reused more if only the type-checker would allow it
  - By using the same data structures for everything (e.g., lists), you can reuse lots of libraries

- Static:
  - Using separate types catches bugs and enforces abstractions (don't accidentally confuse two lists)
  - Advanced types can provide enough flexibility in practice

Whether to encode with an existing type and use libraries or make a new type is a key design trade-off

Exploring some arguments

6. Types make programs slower/faster

- Static
  - Faster and smaller because programmer controls where tag tests occur and which tags are actually stored
    - Example: “Only when using datatypes”

- Dynamic:
  - Faster because don’t have to code around the type system
  - Optimizer can remove [some] unnecessary tag tests [and tends to do better in inner loops]

Exploring some arguments

7. (a) Dynamic better for prototyping

Early on, you may not know what cases you need in datatypes and functions

- But static typing disallows code without having all cases; dynamic lets incomplete programs run
- So you make premature commitments to data structures
- And end up writing code to appease the type-checker that you later throw away
  - Particularly frustrating while prototyping

Exploring some arguments

7. (b) Static better for prototyping

What better way to document your evolving decisions on data structures and code-cases than with the type system?

- New, evolving code most likely to make inconsistent assumptions

Easy to put in temporary stubs as necessary, such as

```
| _ -> raise Unimplemented
```

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 *programming language*
  - Define a *type system* to rule out programs we don't want
- What logicians do
  - Define a *logic* (a way to state propositions)
    - E.g.,: **`f ::= p | f or f | f and f | f -> f`**
  - Define a *proof system* (a [sound] way to prove propositions)
- It turns out we did that too!

- Slogans:
  - "Propositions are Types"
  - "Proofs are Programs"

A funny STLC

- Let's take the explicitly typed STLC with:
  - Any number of base types **`b1`**, **`b2`**, …
  - pairs
  - sums
  - no constants (can add one or more if you want)

Expressions: e ::= x | λx:**`τ`**. e | e e | (e,e) | e.1 | e.2
| A e | B e | match e with A x**->**e |B x**->**e

Types: **`τ ::= b1|b2|… | τ→ τ | τ*τ | τ+τ`**

Even without constants, plenty of terms type-check with Γ = .

Example programs

λ**x:b17. x**

has type

b17 $\rightarrow$ **b17**

Example programs

λ**x:b1. λf:b1→b2. f x**

has type

b1 → (b1 → b2) → b2

Example programs

`λx:b1→b2→b3. λy:b2. λz:b1. x z y`

has type

`(b1 → b2 → b3) → b2 → b1 → b3`

Example programs

```
λx:b1. (A(x), A(x))
```

has type

```
b1 → ((b1+b7) * (b1+b4))
```

Example programs

```
λf:b1→b3. λg:b2→b3. λz:b1+b2.
  (match z with A x. f x | B x. g x)
```

has type

```
(b1 → b3) → (b2 → b3) → (b1 + b2) → b3
```

Example programs

λ`x:b1*b2.` λ`y:b3. ((y,x.1),x.2)`

has type

`(b1*b2)` $\rightarrow$ `b3` $\rightarrow$ `((b3*b1)*b2)`

Empty and nonempty types

So we have types for which there are closed values:

```
b17 → b17
b1 → (b1 → b2) → b2
(b1 → b2 → b3) → b2 → b1 → b3
b1 → ((b1+b7) * (b1+b4))
(b1 → b3) → (b2 → b3) → (b1 + b2) → b3
(b1*b2) → b3 → ((b3*b1)*b2)
```

But there are also many types for which there are no closed values:

```
b1     b1→b2     b1+(b1→b2)     b1→(b2→b1)→b2
```

And "I" have a "secret" way of knowing which types have values

- Let me show you propositional logic…

Propositional Logic

With → for implies, **+** for inclusive-or and ***** for and:

$$p ::= p1 \mid p2 \mid \ldots \mid p \to p \mid p * p \mid p + p$$

$$\Gamma ::= . \mid \Gamma, p$$

$$\boxed{\Gamma \vdash p}$$

$$\frac{\Gamma \vdash p1 \quad \Gamma \vdash p2}{\Gamma \vdash p1 * p2} \qquad \frac{\Gamma \vdash p1 * p2}{\Gamma \vdash p1} \qquad \frac{\Gamma \vdash p1 * p2}{\Gamma \vdash p2}$$

$$\frac{\Gamma \vdash p1}{\Gamma \vdash p1 + p2} \qquad \frac{\Gamma \vdash p2}{\Gamma \vdash p1 + p2} \qquad \frac{\Gamma \vdash p1 + p2 \quad \Gamma, p1 \vdash p3 \quad \Gamma, p2 \vdash p3}{\Gamma \vdash p3}$$

$$\frac{p \text{ in } \Gamma}{\Gamma \vdash p} \qquad \frac{\Gamma, p1 \vdash p2}{\Gamma \vdash p1 \to p2} \qquad \frac{\Gamma \vdash p1 \to p2 \quad \Gamma \vdash p1}{\Gamma \vdash p2}$$

Guess what!!!

That's exactly our type system, just:

- Erasing terms
- Changing every **τ** to a **p**

So our type system ***is*** a proof system for propositional logic

- Function-call rule is modus ponens
- Function-definition rule is implication-introduction
- Variable-lookup rule is assumption
- **e**.1 and **e**.2 rules are and-elimination
- …

Curry-Howard Isomorphism

- Given a closed term that type-checks, take the typing derivation, erase the terms, and have a propositional-logic proof
- Given a propositional-logic proof of a formula, there exists a closed lambda-calculus term with that formula for its type *(almost)*
- A term that type-checks is a *proof* – it tells you exactly how to derive the logic formula corresponding to its type
- Lambdas are no more or less made up than logical implication!
  - STLC with pairs and sums *is* [constructive] propositional logic
- Let's revisit our examples under the logical interpretation…

Example programs

λ**`x:b17. x`**

is a proof that

`b17` $\rightarrow$ **`b17`**

Example programs

`λx:b1. λf:b1→b2. f x`

is a proof that

`b1 → (b1 → b2) → b2`

Example programs

λ**x:b1→b2→b3.** λ**y:b2.** λ**z:b1. x z y**

is a proof that

(b1 → b2 → b3) → b2 → b1 → b3

Example programs

$$\lambda x{:}b1.\ (A(x),\ A(x))$$

is a proof that

$$b1 \rightarrow ((b1+b7) * (b1+b4))$$

Example programs

```
λf:b1→b3. λg:b2→b3. λz:b1+b2.
  (match z with A x. f x | B x. g x)
```

is a proof that

```
(b1 → b3) → (b2 → b3) → (b1 + b2) → b3
```

Example programs

```
λx:b1*b2. λy:b3. ((y,x.1),x.2)
```

is a proof that

```
(b1*b2) → b3 → ((b3*b1)*b2)
```

Why care?

- Makes me glad I'm not a dog
- Don't think of logic and computing as distinct fields
- Thinking "the other way" can help you debug interfaces
- Type systems are not *ad hoc* piles of rules!

STLC is a *sound* proof system for propositional logic

- But it's not quite *complete*…

Classical vs. Constructive

Classical propositional logic has the "law of the excluded middle":

$$\frac{}{\Gamma \vdash \mathtt{p1+(p1 \rightarrow p2)}}$$

Think "p or not p" or double negation (we don't have a not)

Logics without this rule (or anything equivalent) are called *constructive.* They're useful because proofs "know how the world is" and therefore "are executable."

Our match rule let's us "branch on possibilities", but *using it* requires *knowing* which possibility holds [or that both do]:

$$\frac{\Gamma \vdash \mathtt{p1+p2} \quad \Gamma\mathtt{,p1} \vdash \mathtt{p3} \quad \Gamma\mathtt{,p2} \vdash \mathtt{p3}}{\Gamma \vdash \mathtt{p3}}$$

Example classical proof

Theorem: I can always wake up at 9 and be at work by 10.

Proof: If it's a weekday, I can take a bus that leaves at 9:30. If it is not a weekday, traffic is light and I can drive. *Since it is a weekday or it is not a weekday*, I can be at work by 10.

Problem: If you wake up and don't know if it's a weekday, this proof does not let you construct a plan to get to work by 10.

In constructive logic, if a theorem is proven, we have a plan/program

- And you can still prove, "If I know whether or not it is a weekday, then I can wake up at 9 and be at work by 10"

What about recursion

- letrec lets you prove anything
  - (that's bad – an "inconsistent logic")

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

- Only terminating programs are proofs!

- Related: In ML, a function of type **`int → 'a`** never returns normally

Last word on Curry-Howard

- It's not just STLC and constructive propositional logic
  - Every logic has a corresponding typed lambda calculus and vice-versa
  - Generics correspond to universal quantification

- If you remember one thing: the typing rule for function application is implication-elimination (a.k.a. modus ponens)