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 (∀), Abstract types (∃)
- 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:  \( \tau ::= \text{int} \mid \tau \rightarrow \tau \)

Contexts:  \( \Gamma ::= . \mid \Gamma, x : \tau \)

\[
\begin{align*}
\text{e} &\to \text{e}' \\
\text{e}_1 \text{e}_2 &\to \text{e}_1' \text{e}_2 \\
\text{v} \text{e}_2 &\to \text{v} \text{e}_2' \\
(\lambda \text{x}. \text{e}) \text{v} &\to \text{e}\{\text{v/x}\}
\end{align*}
\]
Rule-by-rule

- Constant rule: context irrelevant
- Variable rule: lookup (no instantiation if $x$ not in $\Gamma$)
- 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 \( \tau \)
- If we think of \( \Gamma \) 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)
    
    \[
    (\lambda x. (x (\lambda y. y)) (x \ 3)) \ (\lambda z. z)
    \]
Sound

• Statement of soundness theorem:
  If \( \vdash e : \tau \) and \( e \rightarrow^* e_2 \),
  then \( e_2 \) is a value or there exists an \( e_3 \) such that \( e_2 \rightarrow e_3 \)

• Proof is non-trivial
  – Must hold for all \( e \) and any number of steps
  – But easy given two helper theorems…

  1. Progress: If \( \vdash e : \tau \), then \( e \) is a value or there exists an \( e_2 \) such that \( e \rightarrow e_2 \)

  2. Preservation: If \( \vdash e : \tau \) and \( e \rightarrow e_2 \), then \( \vdash e_2 : \tau \)
Let’s prove it

Prove: If \( \vdash e : \tau \) and \( e \rightarrow^* e_2 \),
then \( e_2 \) is a value or \( \exists e_3 \) such that \( e_2 \rightarrow e_3 \), assuming:

1. If \( \vdash e : \tau \) then \( e \) is a value or \( \exists e_2 \) such that \( e \rightarrow e_2 \)
2. If \( \vdash e : \tau \) and \( e \rightarrow e_2 \) then \( \vdash e_2 : \tau \)

Prove something stronger: Also show \( \vdash e_2 : \tau \)

Proof: By induction on \( n \) where \( e \rightarrow^* e_2 \) in \( n \) steps

• Case \( n=0 \): immediate from progress (\( e=e_2 \))
• Case \( n>0 \): then \( \exists e_3 \) 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 $\vdash e : \tau$ then $e$ is a value or there exists an $e_2$ such that $e \rightarrow e_2$

Proof: Induction on height of derivation tree for $\vdash e : \tau$

Rough idea:
- Trivial unless $e$ is an application
- For $e = e_1 \ e_2$,
  - If left or right not a value, induction
  - If both values, $e_1$ must be a lambda...
Forget a couple things?

Preservation: If $\vdash e : \tau$ and $e \rightarrow e_2$ then $\vdash e_2 : \tau$

Also by induction on assumed typing derivation.

The trouble is when $e \rightarrow e'$ involves substitution
  
  – Requires another theorem

Substitution:
  
  If $\Gamma, x : \tau_1 \vdash e : \tau$ and $\Gamma \vdash e_1 : \tau_1$,
  then $\Gamma \vdash e\{e_1/x\} : \tau$
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 (∀), Abstract types (∃), 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)

\[ e ::= \ldots \mid \text{let } x = e_1 \text{ in } e_2 \]

(no new values or types)

\[ e_1 \to e_1' \]

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

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

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

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

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

2. \( \text{let } x = e_1 \text{ in } e_2 \) is sugar (a different concrete way to write the same abstract syntax) for \((\lambda x. e_2) \ e_1\)

3. Instead of rules on last slide, just use

\[
\text{let } x = e_1 \text{ in } e_2 \rightarrow (\lambda x. e_2) \ e_1
\]

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

\[ e ::= \ldots \mid \text{tru} \mid \text{fls} \mid e \ ? \ e : e \]
\[ v ::= \ldots \mid \text{tru} \mid \text{fls} \]
\[ \tau ::= \ldots \mid \text{bool} \]

\[ e_1 \rightarrow e_1' \]

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

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

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

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

\[
e ::= \ldots \mid (e, e) \mid e.1 \mid e.2
\]

\[
v ::= \ldots \mid (v, v)
\]

\[
\tau ::= \ldots \mid \tau \tau
\]

\[
\begin{align*}
e_1 \rightarrow & e_1' \\
(e_1, e_2) \rightarrow & (e_1', e_2) \\
(v, e_2) \rightarrow & (v, e_2') \\
e.1 \rightarrow & e'.1 \\
e.2 \rightarrow & e'.2 \\
(v_1, v_2).1 \rightarrow & v_1 \\
(v_1, v_2).2 \rightarrow & v_2
\end{align*}
\]

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

\[
\begin{align*}
\Gamma \vdash & e : \tau_1 \tau_2 \\
\Gamma \vdash & e.1 : \tau_1 \\
\Gamma \vdash & e.2 : \tau_2
\end{align*}
\]
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 expression
  - Elimination forms: `match e1 with pat -> exp ...`
  - Typing: If \( e \) has type \( t_1 \), 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)

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

\[
\begin{align*}
  e & \rightarrow e' \quad e & \rightarrow e' \quad e1 & \rightarrow e1' \\
  A \ e & \rightarrow A \ e' \quad B \ e & \rightarrow B \ e' \quad \text{match } e1 \text{ with } A \ x\to e2 \mid B \ y \to e3 \\
  & \quad \rightarrow \text{match } e1' \text{ with } A \ x\to e2 \mid B \ y \to e3
\end{align*}
\]

\[
\begin{align*}
  \text{match } A \ v \text{ with } A \ x\to e2 \mid B \ y \to e3 & \rightarrow e2\{v/x\} \\
  \text{match } B \ v \text{ with } A \ x\to e2 \mid B \ y \to e3 & \rightarrow e3\{v/y\}
\end{align*}
\]
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 OCaml as in Lecture 1
- Example values of type int + (int -> int):

\[
\begin{array}{cc}
0 & 17 \\
1 & \lambda x. x+y [(“y”,6)]
\end{array}
\]
Typing rules

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

\[
\begin{align*}
\Gamma & \vdash e : \tau_1 \\
\Gamma & \vdash A \ e : \tau_1+\tau_2 \\
\Gamma & \vdash B \ e : \tau_1+\tau_2 \\
\Gamma & \vdash e_1 : \tau_1+\tau_2 \\
\Gamma , x : \tau_1 & \vdash e_2 : \tau \\
\Gamma , y : \tau_2 & \vdash e_3 : \tau \\
\hline
\Gamma & \vdash \text{match } e_1 \text{ with } A \ x \rightarrow e_2 \ | B \ y \rightarrow e_3 : \tau
\end{align*}
\]
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”

\[
\begin{align*}
\Gamma & \vdash e_1 : \tau_1 \\
\Gamma & \vdash e_2 : \tau_2 \\
\hline
\Gamma & \vdash (e_1, e_2) : \tau_1 \times \tau_2 \\
\Gamma & \vdash A \ e : \tau_1 + \tau_2 \\
\Gamma & \vdash B \ e : \tau_1 + \tau_2
\end{align*}
\]
Compare to pairs, part 2

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

\[
\begin{align*}
\Gamma \vdash e : \tau_1 \times \tau_2 & \quad \Gamma \vdash e : \tau_1 \times \tau_2 \\
\hline
\Gamma \vdash e.1 : \tau_1 & \quad \Gamma \vdash e.2 : \tau_2
\end{align*}
\]

\[
\begin{align*}
\Gamma \vdash e_1 : \tau_1 + \tau_2 & \quad \Gamma, x : \tau_1 \vdash e_2 : \tau & \quad \Gamma, y : \tau_2 \vdash e_3 : \tau \\
\hline
\Gamma \vdash \text{match } e_1 \text{ with } A \ x \rightarrow e_2 \mid B \ y \rightarrow e_3 : \tau
\end{align*}
\]
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: \( \text{int} + (\text{int} \to \text{int}) \) becomes
    \[
    \text{int} \times (\text{int} \times (\text{int} \to \text{int}))
    \]
Sums in other guises

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

match e with A x -> ...

Meets C:

```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

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

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

Meets Java [C# similar]:

```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
  - New constructors vs. new operations over the datatype!
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 $\vdash e : \tau$ then there exists a value $v$ such that $e \rightarrow^* v$
  - 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

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

(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 (∃)
• 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 $+\infty.0$!
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 *)
     
     ```ml
     let f x = if x>0 then 2*x else false
     ...
     let ans = (f 19) + 4
     
     versus
     (*) actual OCaml *)
     ```
     
     ```ml
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 *)

```ocaml
let cube x = if int? x
  then x*x*x
  else raise Failure
```

versus

(* actual OCaml *)

```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 type-checker”

Example: currying versus tupling:

(* does not type-check *)

```plaintext
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 *)

```ml
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: \[
\text{let } \text{cube } x = x \times x \times x
\]

To: \[
\text{type } t = \text{I of int} \mid \text{S of string}
\]
\[
\text{let } \text{cube } x = \text{match } x \text{ with } \text{I } i \rightarrow i \times i \times i
\]
\[
\mid \text{S } s \rightarrow s \times s \times s
\]

– Static: Must change all existing callers

Dynamic: No change to existing callers…
\[
\text{let } \text{cube } x = \text{if int? } x \text{ then } x \times x \times x
\]
\[
\text{else } x \times x \times x
\]
Exploring some arguments

4. (b) “Static typing better for code evolution”

Imagine changing the return type instead of the argument type:

```haskell
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

```haskell
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., $f ::= p \mid f \text{ or } f \mid f \text{ and } f \mid f \rightarrow f$
  – 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 \( b_1, b_2, \ldots \)
  - pairs
  - sums
  - no constants (can add one or more if you want)

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

Types: \( \tau ::= b_1 | b_2 | \ldots \mid \tau \rightarrow \tau \mid \tau \tau \mid \tau + \tau \)

Even without constants, plenty of terms type-check with \( \Gamma = . \)
Example programs

\[ \lambda x : b17. \ x \]

has type

\[ b17 \rightarrow b17 \]
Example programs

\[ \lambda x : b_1. \ \lambda f : b_1 \to b_2. \ f \ x \]

has type

\[ b_1 \to (b_1 \to b_2) \to b_2 \]
Example programs

\[ \lambda x : b_1 \to b_2 \to b_3. \; \lambda y : b_2. \; \lambda z : b_1. \; x \; z \; y \]

has type

\[ (b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3 \]
Example programs

\[ \lambda x : b1. \ (A(x), \ A(x)) \]

has type

\[ b1 \to ((b1+b7) \times (b1+b4)) \]
Example programs

\[ \lambda f : b_1 \rightarrow b_3. \lambda g : b_2 \rightarrow b_3. \lambda z : b_1 + b_2. \]

\[
\text{(match } z \text{ with } A \ x. \ f \ x \ | \ B \ x. \ g \ x)\]

has type

\[
(b_1 \rightarrow b_3) \rightarrow (b_2 \rightarrow b_3) \rightarrow (b_1 + b_2) \rightarrow b_3
\]
Example programs

\( \lambda x : b_1 * b_2. \; \lambda y : b_3. \; ((y, x.1), x.2) \)

has type

\((b_1 * b_2) \to b_3 \to ((b_3 * b_1) * b_2)\)
Empty and nonempty types

So we have types for which there are closed values:

\[ b_1 \rightarrow b_1 \]
\[ b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2 \]
\[ (b_1 \rightarrow b_2 \rightarrow b_3) \rightarrow b_2 \rightarrow b_1 \rightarrow b_3 \]
\[ b_1 \rightarrow ((b_1 + b_7) \ast (b_1 + b_4)) \]
\[ (b_1 \rightarrow b_3) \rightarrow (b_2 \rightarrow b_3) \rightarrow (b_1 + b_2) \rightarrow b_3 \]
\[ (b_1 \ast b_2) \rightarrow b_3 \rightarrow ((b_3 \ast b_1) \ast b_2) \]

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

\[ b_1 \quad b_1 \rightarrow b_2 \quad b_1 + (b_1 \rightarrow b_2) \quad b_1 \rightarrow (b_2 \rightarrow b_1) \rightarrow b_2 \]

And “I” have a “secret” way of knowing which types have values

– Let me show you propositional logic…
Propositional Logic

With $\rightarrow$ for implies, $+$ for inclusive-or and $*$ for and:

$p ::= p_1 | p_2 | \ldots | p \rightarrow p | p*p | p+p$

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

\[
\begin{array}{ll}
\Gamma \vdash p_1 & \Gamma \vdash p_2 \\
\hline
\Gamma \vdash p_1*p_2 & \Gamma \vdash p_1 & \Gamma \vdash p_2 \\
\Gamma \vdash p_1+p_2 & \Gamma \vdash p_1+p_2 & \Gamma \vdash p_3 \\
p \text{ in } \Gamma & \Gamma, p_1 \vdash p_2 & \Gamma \vdash p_1 \rightarrow p_2 \\
\hline
\Gamma \vdash p & \Gamma \vdash p_1 \rightarrow p_2 & \Gamma \vdash p_2
\end{array}
\]
Guess what!!!

That’s exactly our type system, just:
• Erasing terms
• Changing every $\tau$ 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 \textit{(almost)}

- A term that type-checks is a \textit{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 \textit{is} [constructive] propositional logic

- Let’s revisit our examples under the logical interpretation…
Example programs

\[ \lambda x : b17. \ x \]

is a proof that

\[ b17 \rightarrow b17 \]
Example programs

\[ \lambda x : b_1. \lambda f : b_1 \rightarrow b_2. \ f \ x \]

is a proof that

\[ b_1 \rightarrow (b_1 \rightarrow b_2) \rightarrow b_2 \]
Example programs

\( \lambda x : b_1 \to b_2 \to b_3. \ \lambda y : b_2. \ \lambda z : b_1. \ x \ z \ y \)

is a proof that

\( (b_1 \to b_2 \to b_3) \to b_2 \to b_1 \to b_3 \)
Example programs

\( \lambda x : b1. \ (A(x), A(x)) \)

is a proof that

\( b1 \rightarrow ((b1+b7) \ast (b1+b4)) \)
Example programs

\[ \lambda f : b1 \to b3. \ \lambda g : b2 \to b3. \ \lambda z : b1 + b2. \]

\[ (\text{match } z \text{ with } A \ x. \ f \ x \ | \ B \ x. \ g \ x) \]

is a proof that

\[ (b1 \to b3) \to (b2 \to b3) \to (b1 + b2) \to b3 \]
Example programs

\[\lambda x : b1 \ast b2. \lambda y : b3. \ ((y, x.1), x.2)\]

is a proof that

\[(b1 \ast b2) \rightarrow b3 \rightarrow (b3 \ast b1) \ast b2\]
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”:

$$\Gamma \vdash p_1 + (p_1 \rightarrow p_2)$$

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]:

$$\Gamma \vdash p_1 + p_2 \quad \Gamma, p_1 \vdash p_3 \quad \Gamma, p_2 \vdash p_3$$

$$\Gamma \vdash p_3$$
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”)

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

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

- Only terminating programs are proofs!

- Related: In ML, a function of type \texttt{int} \to \texttt{’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)