

Type theory

- Review
 - System F
 - Axiomatic semantics
- Type theory



System F (types)

$t ::=$	X	type variables
	$t_1 \rightarrow t_2$ $t_1 \Rightarrow t_2$	function type
	$t_1 + t_2$ $t_1 \vee t_2$	disjoint union
	$t_1 * t_2$ $t_1 \wedge t_2$	product type
	$\forall X.t$ $\forall X.t$	polymorphism
	$\exists X.t$ $\exists X.t$	abstraction



System F (expressions)

$e ::=$	v	variables
	$\lambda x: t.e$	abstraction
	$e_1 e_2$	application
	$\Lambda X.e$	type abstraction
	$e[t]$	type application
	(e_1, e_2)	pairs
	match e_1 with $(x_1, x_2) \rightarrow e_2$	pair elim
	inl (e)	left injection
	inr (e)	right injection
	match e_1 with \dots	
	$\{t_1, e\}$ as t_2	pack
	let $\{X, x\} = e_1$ in e_2	unpack



System F notes

- Properties
 - Preservation/progress (well-typed programs don't "go wrong")
 - No guarantees about termination (although simple System F is strongly normalizing)
 - Weak Curry-Howard isomorphism (non-termination means *false* is inhabited by the nonterminating program)
- Computability
 - Type inference: undecidable; for Prenex forms: EXPTIME



Total correctness

- For total correctness (the program always terminates with the right answer), use another semantics
 - Axiomatic semantics
 - All programs must terminate
 - Denotational semantics
 - Set-theoretic semantics that allows for non-termination



Axiomatic semantics

```
{b ≥ 0}
x, y, z ← a, b, 0;
do y > 0 ∧ even(y) → y, x ← y/2, x + x
| odd(y) → y, z ← y - 1, z + x
od
{R : z = a * b}
```



Axiomatic semantics

```
{b ≥ 0}
x, y, z ← a, b, 0;
{P}
do  y > 0 ∧ even(y) → {P ∧ y > 0 ∧ even(y)}y, x - y/2, x + x{P}
   | odd(y)          → {P ∧ odd(y)}y, z - y - 1, z + x{P}
od
{P ∧ y ≤ 0 ∧ ¬odd(y)}
{P ∧ y = 0}
{R : z = a * b}
```



Axiomatic semantics

- Total correctness
 - Programs always terminate with the right answer
 - Progress, not preservation
- Usually imperative languages
 - Can be adapted to functional languages with some effort
- Inference is undecidable
 - Loop invariants
- Non-abstract, non-modular



Type theory

- Increase the expressiveness of the type system



Type theory

- Increase the expressiveness of the type system
 - Use propositions as types principle
 - Generalized syntax (programs can compute types)
- Styles
 - Church-style (what we have been using so far): we only talk about the well-typed programs
 - Curry-style: programs make sense even without a type system: first define the programs, then define type systems to classify the programs
 - We'll use Curry-style, Martin-Lof type theory



Martin-Lof type theory

e, t	::=	•	unit
		v	variables
		$\lambda x.e$	abstraction
		$e_1 e_2$	application
		$\mathbf{inl}(e) \mid \mathbf{inr}(e)$	union
		(e_1, e_2)	pair
		$\cup_1 \mid \cup_2 \mid \dots$	type universe
		\mathbf{Unit}	unit type
		$t_1 + t_2$	union type
		$t_1 * t_2$	product type
		$t_1 \rightarrow t_2$	function type
		$\forall x: t_1. t_2$	dependent function
		$\exists x: t_1. t_2$	dependent product



Type theory

- Abstraction $\lambda x.e$ is untyped
- Quantifications $\forall x: t_1. t_2, \exists x: t_1. t_2$ are bounded
- There is no "type" application $e[t]$ (just normal application)
- The syntax for expressions and types is the same
- Types are classified into "type universes" \cup_1, \cup_2, \dots



Combinators, Booleans

K combinator:

$$(\lambda P. \lambda Q. \lambda x. \lambda y. x) : \forall P: \mathbb{U}_1. \forall Q: \mathbb{U}_1. P \Rightarrow Q \Rightarrow P$$

An encoding of Booleans:

- let $\mathbb{B} \equiv \text{Unit} + \text{Unit}$
- **false** $\equiv \text{inl}(\bullet)$
- **true** $\equiv \text{inr}(\bullet)$



Booleans

· if e_1 then e_2 else e_3 is

$$\text{match } e_1 \text{ with } \text{inl}(_) \rightarrow e_2 \mid \text{inr}(_) \rightarrow e_3$$

$$(\lambda b. \text{if } b \text{ then } 1 \text{ else "foo"}) : ???$$



Dependent function space

$$(\lambda b. \text{if } b \text{ then } 1 \text{ else "foo"}) :$$
$$(\forall b: \mathbb{B}. \text{if } b \text{ then } \mathbb{Z} \text{ else } \text{String})$$



Records

$Joe = \lambda x.$ **if** $x = \text{"name"}$ **then** "Joe"
else if $x = \text{"height"}$ **then** 75
else if $x = \text{"salary"}$ **then** $\$127,000$
else \bullet

$Joe \in \forall x: \text{String}.$ **if** $x = \text{"name"}$ **then** String
else if $x = \text{"height"}$ **then** \mathbb{Z}
else if $x = \text{"salary"}$ **then** \mathbb{R}
else Unit



Group example

$T : \cup_i$
 $+ : (T * T) \rightarrow T$
 $1 : T$
 $inverse : T \rightarrow T$

$Group \equiv$ $T : \cup_i$
 \times $+ : (T * T) \rightarrow T$
 \times $1 : T$
 \times $inverse : T \rightarrow T$

$Int = (\mathbb{N}, \lambda x.(x.1 + x.2), 0, \lambda x.0 - x)$



Typing judgments

- As usual, assume that a judgment is a sequent

$$x_1:t_1, \dots, x_n:t_n \vdash C$$

- Basic judgments:
 - What programs have which types?
 - When are two programs equal?
 - When are two types equal?
 - When is an expression a program?
 - When is an expression a type?



Semantics: pointwise functionality

A sequent $x_1: T_1, \dots, x_n: T_n \vdash C$ means:

Assuming:

- T_1 is a type, and x_1 has type T_1 ,
- and T_2 is a type for any $x_1 \in T_1$, and x_2 has type T_2 ,
- . . .
- and T_n is a type for any $x_1 \in T_1, x_2 \in T_2, \dots$, and x_n has type T_n

Then: the judgment C is true.



Equality semantics

The meaning of $e_1 = e_2 \in T$:

- T is a type
- e_1 and e_2 have type T
- e_1 and e_2 are equal wrt equality in T



Type universes

$$\mathbb{U}_1 \in \mathbb{U}_2 \in \mathbb{U}_3 \in \dots$$

$$\frac{\Gamma \vdash t \in \mathbb{U}_i}{\Gamma \vdash t \in \mathbb{U}_{i+1}} \text{cumulativity}$$

$$\overline{\Gamma \vdash \text{Void} \in \mathbb{U}_1}$$

$$\overline{\Gamma \vdash \text{Unit} \in \mathbb{U}_1}$$

$$\frac{\Gamma \vdash A \in \mathbb{U}_i \quad \Gamma \vdash B \in \mathbb{U}_i}{\Gamma \vdash (A \rightarrow B) \in \mathbb{U}_i}$$



Today

- Elimination rules
 - How do we reason from what we are assuming?
 - Reasoning on the left
- Dependent types



New notation for substitution

Usually, substitution has the form $e_1[e_2/x]$ meaning "substitute e_2 for x in e_1 ."

$$(\lambda x.e_1) e_2 \rightarrow_{\beta} e_2[e_1/x]$$

We'll use the notation for "second-order substitution." The left-side of the rule is a pattern, the right side is a substitution.

$$(\lambda x.e_1[x]) e_2 \rightarrow_{\beta} e_2[e_1]$$



Number rules

$$\frac{}{\Gamma \vdash \mathbb{N} \text{ type}} \text{ nat intro}$$

$$\frac{}{\Gamma \vdash 5 \in \mathbb{N}} \text{ num intro}$$

$$\frac{\Gamma \vdash e_1 \in \mathbb{N} \quad \Gamma \vdash e_2 \in \mathbb{N}}{\Gamma \vdash (e_1 + e_2) \in \mathbb{N}} \text{ plus intro}$$

$$\frac{\Gamma \vdash C[0/x] \quad \Gamma, y:\mathbb{N}; C[y/x] \vdash C[(y+1)/x]}{\Gamma, x:\mathbb{N} \vdash C} \text{ nat elim}$$



Number rules in new form

$$\overline{\Gamma \vdash \mathbb{N} \text{ type}} \text{ nat intro}$$

$$\overline{\Gamma \vdash 5 \in \mathbb{N}} \text{ num intro}$$

$$\frac{\Gamma \vdash e_1 \in \mathbb{N} \quad \Gamma \vdash e_2 \in \mathbb{N}}{\Gamma \vdash (e_1 + e_2) \in \mathbb{N}} \text{ plus intro}$$

$$\frac{\Gamma \vdash C[0] \quad \Gamma, y: \mathbb{N}; C[y] \vdash C[(y + 1)]}{\Gamma, x: \mathbb{N} \vdash C[x]} \text{ nat elim}$$



Elimination, in general

- Elimination is
 - A case analysis
 - An induction
- Reasoning forward from what you know

$$\frac{\Gamma \vdash C[0] \quad \Gamma, y: \mathbb{N}; C[y] \vdash C[(y + 1)]}{\Gamma, x: \mathbb{N} \vdash C[x]} \text{ nat elim}$$



Other elimination rules

$$\frac{\Gamma, \Delta[\bullet] \vdash C[\bullet]}{\Gamma, x: \text{Unit}, \Delta[x] \vdash C[x]} \text{ unit elim}$$

$$\frac{\Gamma, x: A, \Delta[\text{inl}(x)] \vdash C[\text{inl}(x)] \quad \Gamma, x: B, \Delta[\text{inr}(x)] \vdash C[\text{inr}(x)]}{\Gamma, x: A + B, \Delta[x] \vdash C[x]} \text{ union elim}$$

$$\frac{\Gamma, u: A, v: B, \Delta[(a, b)] \vdash C[(a, b)]}{\Gamma, x: A * B, \Delta[x] \vdash C[x]} \text{ prod elim}$$

$$\overline{\Gamma, x: \text{Void}, \Delta[x] \vdash C[x]} \text{ void elim}$$



Function elimination

- We can't do induction over *all* the functions
- However, we can apply the function to an argument

$$\frac{\Gamma, f: A \rightarrow B, \Delta[f] \vdash a \in A \quad \Gamma, f: A \rightarrow B, \Delta[f], \gamma: B, \gamma = f(x) \in B \vdash C[f]}{\Gamma, f: A \rightarrow B, \Delta[f] \vdash C[f]} \text{ fun elim}$$



Example: pair elimination

Define an operator for β reduction on pairs:

- $(u, v).1 \rightarrow_{\beta} u$
- $(u, v).2 \rightarrow_{\beta} v$

$$\frac{\frac{\frac{\Gamma, u: A, v: B \vdash u \in A}{\Gamma, u: A, v: B \vdash (u, v).1 \in A} 2}{\Gamma, x: A \times B \vdash x.1 \in A} 1} 3$$



The trivial function space

$$\frac{\frac{\Gamma, x: \text{Void} \vdash (1 + \bullet) \in \text{Unit}}{\Gamma \vdash (\lambda x. 1 + \bullet) \in (\text{Void} \rightarrow \text{Unit})} 1} 2$$

$$\frac{\frac{\Gamma, x: \text{Void} \vdash 1 = 2 \in \mathbb{Z}}{\Gamma \vdash (\lambda x. 1) = (\lambda x. 2) \in (\text{Void} \rightarrow \mathbb{Z})} 1} 2$$



Propositions-as-types

Type	Witness	Interpretation
<i>Void</i>		<i>false</i>
<i>Unit</i>	•	<i>true</i>
$A \times B$	(a, b)	$A \wedge B$
$A + B$	$inl(a), inr(b)$	$A \vee B$
$A \rightarrow B$	$\lambda v. b[v]$	$A \Rightarrow B$
$x: A \times B[x]$	(a, b)	$\exists x: A. B[x]$
$x: A \rightarrow B[x]$	$\lambda v. b[v]$	$\forall x: A. B[x]$



Type formation rules

$$\frac{}{\Gamma \vdash \text{Unit} \text{ ext } \bullet} \text{ unit}$$

$$\frac{\Gamma \vdash A \text{ ext } a \quad \Gamma \vdash B \text{ ext } b}{\Gamma \vdash A \times B \text{ ext } (a, b)} \text{ and}$$

$$\frac{\Gamma \vdash A \text{ ext } a \quad \Gamma \vdash B \text{ type}}{\Gamma \vdash A + B \text{ ext } inl(a)} \text{ or-left}$$

$$\frac{\Gamma, v: A \vdash B \text{ ext } b \quad \Gamma \vdash A \text{ type}}{\Gamma \vdash A \rightarrow B \text{ ext } \lambda v. b} \text{ implies}$$



Proposition formation rules

$$\frac{}{\Gamma \vdash \text{True} \text{ ext } \bullet} \text{ unit}$$

$$\frac{\Gamma \vdash A \text{ ext } a \quad \Gamma \vdash B \text{ ext } b}{\Gamma \vdash A \wedge B \text{ ext } (a, b)} \text{ and}$$

$$\frac{\Gamma \vdash A \text{ ext } a \quad \Gamma \vdash B \text{ type}}{\Gamma \vdash A \vee B \text{ ext } inl(a)} \text{ or-left}$$

$$\frac{\Gamma, v: A \vdash B \text{ ext } b \quad \Gamma \vdash A \text{ type}}{\Gamma \vdash A \Rightarrow B \text{ ext } \lambda v. b} \text{ implies}$$



Quantifier rules

$$\frac{\Gamma \vdash a \in A \quad \Gamma \vdash B[a] \text{ ext } b \quad \Gamma, x: A \vdash B[x] \text{ type}}{\Gamma \vdash \exists x: A. B[x] \text{ ext}(a, b)} \text{ exists}$$

$$\frac{\Gamma, v: A \vdash B[v] \text{ ext } b[v]}{\Gamma \vdash \forall x: A. B[x] \text{ ext } \lambda v. b[v]} \text{ all}$$



An example proof

- Prove that there are an infinite number of natural numbers

$$\frac{\frac{\frac{\Gamma, i: \mathbb{N} \vdash (i+1) > i}{\Gamma, i: \mathbb{N} \vdash \exists j: \mathbb{N}. j > i} 2}{\Gamma \vdash \forall i: \mathbb{N}. \exists j: \mathbb{N}. j > i} 1} 3$$



Example with program extraction

- The program is extracted top-down

$$\frac{\frac{\frac{\Gamma, i: \mathbb{N} \vdash (i+1) > i \text{ ext } \bullet}{\Gamma, i: \mathbb{N} \vdash \exists j: \mathbb{N}. j > i \text{ ext}(i+1, \bullet)} 2}{\Gamma \vdash \forall i: \mathbb{N}. \exists j: \mathbb{N}. j > i \text{ ext } \lambda i. (i+1, \bullet)} 1} 3$$



A more interesting example

- Prove that every natural number has an integer square-root

$$\frac{\text{???}}{\Gamma \vdash \forall i:\mathbb{N}.\exists j:\mathbb{N}.j^2 \leq i \wedge (j+1)^2 > i} \quad 1$$



Square-root, step 1

$$\frac{\text{perform induction}}{\Gamma, i:\mathbb{N} \vdash \exists j:\mathbb{N}.j^2 \leq i \wedge (j+1)^2 > i} \quad 2$$
$$\Gamma \vdash \forall i:\mathbb{N}.\exists j:\mathbb{N}.j^2 \leq i \wedge (j+1)^2 > i \quad 1$$



Square-root, base case

$$\frac{\text{choose } j = 0}{\Gamma \vdash \exists j:\mathbb{N}.j^2 \leq 0 \wedge (j+1)^2 > 0} \quad 3$$



Square-root, induction step

$$\frac{\text{decide whether } (j+1)^2 > (i+1)}{\Gamma, i: \mathbb{N}, j: \mathbb{N}, j^2 \leq i, (j+1)^2 > i \vdash \exists k: \mathbb{N}. k^2 \leq (i+1) \wedge (k+1)^2 > (i+1)} \quad 5$$
$$\Gamma, i: \mathbb{N}, \exists j: \mathbb{N}. j^2 \leq i \wedge (j+1)^2 > i \vdash \exists j: \mathbb{N}. j^2 \leq (i+1) \wedge (j+1)^2 > (i+1) \quad 4$$



Choice 1

choose $k = j$ and use arithmetic 6

$$\Gamma, \\ i: \mathbb{N}, \\ j: \mathbb{N}, \\ j^2 \leq i, \\ (j+1)^2 > i \\ (j+1)^2 > (i+1) \quad \vdash \quad \exists k: \mathbb{N}. k^2 \leq (i+1) \wedge (k+1)^2 > (i+1)$$



Choice 2

choose $k = j + 1$ and use arithmetic 7

$$\Gamma, \\ i: \mathbb{N}, \\ j: \mathbb{N}, \\ j^2 \leq i, \\ (j+1)^2 > i \\ (j+1)^2 = (i+1) \quad \vdash \quad \exists k: \mathbb{N}. k^2 \leq (i+1) \wedge (k+1)^2 > (i+1)$$



Program

```
let rec sqrt i =  
  if i = 0 then  
    0  
  else  
    let j = sqrt (i - 1) in  
      if (j + 1)2 = i + 1 then  
        j + 1  
      else  
        j
```


