# Type Systems

Lecture 14
ECS 240

## Review

- $\lambda$-calculus is as expressive as a Turing machine
- We can encode a multitude of data types in the untyped $\lambda$-calculus
- To simplify programming it is useful to add types to the language
- We now start the study of type systems in the context of the typed $\lambda$-calculus

# Types

- A program variable can assume a range of values during the execution of a program
- An upper bound of such a range is called a <u>type</u> of the variable
  - A variable of type "bool" should only assume boolean values
  - If x has type "bool" then
    - "not(x)" has a sensible meaning
    - but "1 + x" should not be allowed

# Typed and Untyped Languages

- Untyped languages
  - Do not restrict the range of values for a given variable
  - Operations might be applied to inappropriate arguments. The behavior in such cases might be unspecified
  - The pure λ-calculus is an extreme case of an untyped language (however, its behavior is completely specified)

- Typed languages
  - Variables are assigned (non-trivial) types
  - A type system keeps track of types
  - Types might or might not appear in the program itself
  - Languages can be <u>explicitly typed</u> or <u>implicitly typed</u>

# Execution Errors

- The purpose of types is to prevent certain types of execution errors

- Trapped execution errors
  - Cause the computation to stop immediately
  - Well-specified behavior
  - Usually enforced by hardware
  - E.g., Division by zero
  - E.g., Invoking a floating point operation with a NaN
  - E.g., Dereferencing the address 0

## Execution Errors (II)

- Untrapped execution errors
  - Behavior is unspecified (depends on the state of the machine)
  - Accessing past the end of an array
  - Jumping to an address in the data segment
- A program is considered safe if it does not cause untrapped errors
  - Languages in which all programs are safe are <u>safe languages</u>
- For a given language designate a set of forbidden errors
  - A superset of the untrapped errors
  - Includes some trapped errors as well
    - E.g., null pointer dereference
    - To ensure portability across architectures

## Preventing Forbidden Errors - *Static Checking*

- Forbidden errors can be caught by a combination of static and run-time checking

- Static checking
  - Detects errors early, before testing
  - Types provide the necessary static information for static checking
  - E.g., ML, Modula-3, Java
  - Detecting certain/most errors statically is undecidable in most languages

## Preventing Forbidden Errors - *Dynamic Checking*

- Required when static checking is undecidable
  - e.g., array-bounds checking
- Run-time encoding of types are still used
  - e.g., Scheme, Lisp
- Should be limited
  - Delays the manifestation of errors
- Can be done in hardware
  - e.g. null-pointer

# Safe Languages

- There are typed languages that are not safe (weakly typed languages)

- All safe languages use types (either statically or dynamically)

| | Typed | | Untyped |
|---|---|---|---|
| | Static | Dynamic | |
| Safe | ML, Java, ... | Lisp, Scheme | λ-calculus |
| Unsafe | C, C++, ... | ? | Assembly |

- We will be concerned mainly with statically typed languages

# Why Typed Languages?

- Development
  - Type checking catches many mistakes early
  - Reduced debugging time
  - Typed signatures are a powerful basis for design
  - Typed signatures enable separate compilation
- Maintenance
  - Types act as checked specifications
  - Types can enforce abstraction
- Execution
  - Static checking reduces the need for dynamic checking
  - Safe languages are easier to analyze statically
    - the compiler can generate better code

# Why Not Typed Languages?

- Static type checking imposes constraints on the programmer
  - Some valid programs might be rejected
  - But often they can be made well-typed easily
  - Hard to step outside the language (e.g. OO programming in a non-OO language)

- Dynamic safety checks can be costly
  - 50% is a possible cost of bounds-checking in a tight loop
    - In practice, the overall cost is much smaller
  - Memory management must be automatic $\Rightarrow$ need a garbage collector with the associated run-time costs
  - Some applications are justified to use weakly-typed languages

## Properties of Type Systems

- How do types differ from other program annotations
  - Types are more precise than comments
  - Types are more easily mechanizable than program specifications
- Expected properties of type systems:
  - Types should be enforceable
  - Types should be checkable algorithmically
  - Typing rules should be transparent
    - It should be easy to see why a program is not well-typed

## Why Formal Type Systems?

- Many typed languages have informal descriptions of the type systems (e.g., in language reference manuals)
- A fair amount of careful analysis is required to avoid false claims of type safety
- A formal presentation of a type system is a precise specification of the type checker
  - And allows formal proofs of type safety
- But even informal knowledge of the principles of type systems help

# Formalizing a Type System

A multi-step process

1. Syntax
   - Of expressions (programs)
   - Of types
   - Issues of binding and scoping
2. Static semantics (typing rules)
   - Define the typing judgment and its derivation rules
3. Dynamic semantics (e.g., operational)
   - Define the evaluation judgment and its derivation rules
4. Type soundness
   - Relates the static and dynamic semantics
   - State and prove the soundness theorem

# Typing Judgments

- Judgments
  - A statement J about certain formal entities
  - Has a truth value $\models J$
  - Has a derivation $\vdash J$
- A common form of the typing judgment: $\Gamma \vdash e : \tau$
  (e is an expression and $\tau$ is a type)
- $\Gamma$ is a set of type assignments for the free variables of e
  - Defined by the grammar $\Gamma ::= \cdot \mid \Gamma, x : \tau$
  - Usually viewed as a set of type assignments
  - Type assignments for variables not free in e are not relevant
  - E.g, $x : int, y : int \vdash x + y : int$

## Typing rules

- Typing rules are used to derive typing judgments
- Examples:

$$\frac{}{\Gamma \vdash 1 : \texttt{int}}$$

$$\frac{x : \tau \ \in \ \Gamma}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

## Typing Derivations

- A typing derivation is a derivation of a typing judgment
- Example:

$$\dfrac{\dfrac{}{x:\mathtt{int} \vdash x:\mathtt{int}} \qquad \dfrac{\dfrac{}{x:\mathtt{int} \vdash x:\mathtt{int}} \quad \dfrac{}{x:\mathtt{int} \vdash 1:\mathtt{int}}}{x:\mathtt{int} \vdash x+1:\mathtt{int}}}{x:\mathtt{int} \vdash x+(x+1):\mathtt{int}}$$

- We say that $\Gamma \vdash e : \tau$ to denote that there is a derivation of this typing judgment
- Type checking: given $\Gamma$, e and $\tau$ find a derivation
- Type inference: given $\Gamma$ and e, find $\tau$ and a derivation

## Proving Type Soundness

- A typing judgment has a truth value
- Define what it means for a value to have a type

  $v \in \| \tau \|$

  (e.g. $5 \in \| \text{int} \|$ and $\text{true} \in \| \text{bool} \|$ )
- Define what it means for an expression to have a type

  $e \in | \tau |$ iff $\forall v. (e \Downarrow v \Rightarrow v \in \| \tau \|)$
- Prove type soundness

  If $\cdot \vdash e : \tau$ then $e \in | \tau |$

  or equivalently

  If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $v \in \| \tau \|$
- This implies safe execution (since the result of an unsafe execution is not in $\| \tau \|$ for any $\tau$)

# Next

- We will give formal description of first-order type systems (no type variables)
  - Function types (simply typed $\lambda$-calculus)
  - Simple types (integers and booleans)
  - Structured types (products and sums)
  - Imperative types (references and exceptions)
  - Recursive types

- The type systems of most common languages are first-order

- The we move to second-order type systems
  - Polymorphism and abstract types

# First-Order Type Systems

# Simply-Typed Lambda Calculus

- Syntax:

  Terms $e ::= x \mid \lambda x{:}\tau.\ e \mid e_1\ e_2$

  $\mid\ n \mid e_1 + e_2 \mid \text{iszero } e$

  $\mid \text{true} \mid \text{false} \mid \text{not } e \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3$

  Types $\tau ::= \text{int} \mid \text{bool} \mid \tau_1 \to \tau_2$
- $\tau_1 \to \tau_2$ is the function type
- $\to$ associates to the right
- Arguments have typing annotations
- This language is also called $F_1$

## Static Semantics of $F_1$

- The typing judgment

$$\Gamma \vdash e : \tau$$

- The typing rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau.e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}$$

# Static Semantics of $F_1$ (Cont.)

- More typing rules

$$\frac{}{\Gamma \vdash n : \texttt{int}} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}$$

$$\frac{}{\Gamma \vdash \texttt{true} : \texttt{bool}} \qquad \frac{\Gamma \vdash e : \texttt{bool}}{\Gamma \vdash \texttt{not}\ e : \texttt{bool}}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \texttt{if}\ e_1\ \texttt{then}\ e_t\ \texttt{else}\ e_f : \tau}$$

# Typing Derivation in $F_1$

- Consider the term

  λx : int. λb : bool. if b then f x else x

  – With the initial typing assignment f : int → int

$$
\dfrac{\dfrac{}{\Gamma \vdash b : \texttt{bool}} \quad \dfrac{\Gamma \vdash f : \texttt{int} \rightarrow \texttt{int} \quad \Gamma \vdash x : \texttt{int}}{\Gamma \vdash f\ x : \texttt{int}} \quad \Gamma \vdash x : \texttt{int}}{\dfrac{f : \texttt{int} \rightarrow \texttt{int}, x : \texttt{int}, b : \texttt{bool} \vdash \texttt{if}\ b\ \texttt{then}\ f\ x\ \texttt{else}\ x : \texttt{int}}{\dfrac{f : \texttt{int} \rightarrow \texttt{int}, x : \texttt{int} \vdash \lambda b : \texttt{bool}.\ \texttt{if}\ b\ \texttt{then}\ f\ x\ \texttt{else}\ x : \texttt{bool} \rightarrow \texttt{int}}{f : \texttt{int} \rightarrow \texttt{int} \vdash \lambda x : \texttt{int}.\lambda b : \texttt{bool}.\ \texttt{if}\ b\ \texttt{then}\ f\ x\ \texttt{else}\ x : \texttt{int} \rightarrow \texttt{bool} \rightarrow \texttt{int}}}}
$$

Where Γ = f : int → int, x : int, b : bool

# Type Checking in $F_1$

- Type checking is easy because
  - Typing rules are syntax directed
  - Typing rules are compositional
  - All local variables are annotated with types

- In fact, type inference is also easy for $F_1$

- Without type annotations an expression does not have a unique type

$$\cdot \vdash \lambda x.\ x : \text{int} \to \text{int}$$

$$\cdot \vdash \lambda x.\ x : \text{bool} \to \text{bool}$$

# Operational Semantics of $F_1$

- Judgment:

$$e \Downarrow v$$

- Values

$$v ::= n \mid true \mid false \mid \lambda x{:}\tau.\ e$$

- The evaluation rules ...

# Operational Semantics of $F_1$ (Cont.)

- Call-by-value evaluation rules (sample)

$$\frac{}{\lambda x:\tau.e \Downarrow \lambda x:\tau.e}$$

$$\frac{e_1 \Downarrow \lambda x:\tau.e_1' \quad e_2 \Downarrow v_2 \quad [v_2/x]e_1' \Downarrow v}{e_1\ e_2 \Downarrow v}$$

$$\frac{}{n \Downarrow n} \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2 \quad n = n_1 + n_2}{e_1 + e_2 \Downarrow n}$$

$$\frac{e_1 \Downarrow \texttt{true} \quad e_t \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_t\ \texttt{else}\ e_f \Downarrow v}$$

$$\frac{e_1 \Downarrow \texttt{false} \quad e_f \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_t\ \texttt{else}\ e_f \Downarrow v}$$

Evaluation undefined for ill-typed programs !

## Type Soundness for $F_1$

- Theorem:

  If $\cdot \vdash e : \tau$ and $e \Downarrow v$ then $\cdot \vdash v : \tau$

  - Also called, <u>subject reduction</u> theorem, <u>type preservation</u> theorem
- Try to prove by induction on e
  - Won't work because $[v_2/x]e'_1$ in the evaluation of $e_1\ e_2$
  - Same problem with induction on $\cdot \vdash e : \tau$
- Try to prove by induction on $\tau$
  - Won't work because $e_1$ has a "bigger" type than $e_1\ e_2$
- Try to prove by induction on $e \Downarrow v$
  - To address the issue of $[v_2/x]e'_1$
  - This is it!

## Type Soundness Proof

- Consider the case

$$\mathcal{E} :: \frac{e_1 \Downarrow \lambda x : \tau_2 . e_1' \quad e_2 \Downarrow v_2 \quad [v_2/x]e_1' \Downarrow v}{e_1\ e_2 \Downarrow v}$$

and by inversion on the derivation of $e_1\ e_2 : \tau$

$$\mathcal{D} :: \frac{\cdot \vdash e_1 : \tau_2 \longrightarrow \tau \quad \cdot \vdash e_2 : \tau_2}{\cdot \vdash e_1\ e_2 : \tau}$$

- From IH on $e_1 \Downarrow \ldots$ we have $\cdot, x : \tau_2 \vdash e_1' : \tau$
- From IH on $e_2 \Downarrow \ldots$ we have $\cdot \vdash v_2 : \tau_2$
- Need to infer that $\cdot \vdash [v_2/x]e_1' : \tau$ and use the IH
  - We need a substitution lemma (by induction on $e_1'$)

## Significance of Type Soundness

- The theorem says that the result of an evaluation has the same type as the initial expression
- The theorem <u>does not</u> say that
  - The evaluation never gets stuck (e.g., trying to apply a non-function, to add non-integers, etc.), nor that
  - The evaluation terminates

- Even though both of the above facts are true of $F_1$

- We need a small-step semantics to prove that the execution never gets stuck

# Small-Step Contextual Semantics for $F_1$

- We define redexes

  $r ::= n_1 + n_2 \mid \text{if } b \text{ then } e_1 \text{ else } e_2 \mid (\lambda x{:}\tau.e_1)\ v_2$

- and contexts

  $H ::= H_1 + e_2 \mid n_1 + H_2 \mid \text{if } H \text{ then } e_1 \text{ else } e_2 \mid H_1\ e_2 \mid (\lambda x{:}\tau.\ e_1)\ H_2$

- and local reduction rules

  $n_1 + n_2 \rightarrow n_1 \text{ plus } n_2$

  $\text{if true then } e_1 \text{ else } e_2 \rightarrow e_1$

  $\text{if false then } e_1 \text{ else } e_2 \rightarrow e_2$

  $(\lambda x{:}\tau.\ e_1)\ v_2 \rightarrow [v_2/x]e_1$

- and one global reduction rule

  $H[r] \rightarrow H[e] \quad \text{iff } r \rightarrow e$

## Contextual Semantics for $F_1$

- Decomposition lemmas:
  1. If $\cdot \vdash e : \tau$ and e is not a value then there exist (unique) H and r such that e = H[r]
     - any well typed expression can be decomposed
     - Any well-typed non-value can make progress
  2. Furthermore, there exists $\tau'$ such that $\cdot \vdash r : \tau'$
     - the redex is closed and well typed

  3. Furthermore, there exists e' such that $r \to e'$ and $\cdot \vdash e' : \tau'$
     - local reduction is type preserving

  4. Furthermore, for any e' , $\cdot \vdash e' : \tau'$ implies $\cdot \vdash H[e'] : \tau$
     - the expression preserves its type if we replace the redex with an expression of same type

## Contextual Semantics of $F_1$

- Type preservation theorem
  - If $\cdot \vdash e : \tau$ and $e \to e'$ then $\cdot \vdash e' : \tau$
  - Follows from the decomposition lemma
- Progress theorem
  - If $\cdot \vdash e : \tau$ and e is not a value then there exists e' such that e can make progress: $e \to e'$
- Progress theorem says that execution can make progress on a well typed expression
- Furthermore, due to type preservation we know that the execution of a well typed expression never gets stuck
  - this is a common way to state and prove type safety of a language

## Product Types - Static Semantics

- Extend the syntax with (binary) tuples

  $e ::= \ldots \mid (e_1, e_2) \mid \text{fst } e \mid \text{snd } e$

  $\tau ::= \ldots \mid \tau_1 \times \tau_2$

  - This language is sometimes called $F_1^{\times}$
- Same typing judgment $\Gamma \vdash e : \tau$

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash (e_1, e_2) : \tau_1 \times \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{fst}\ e : \tau_1} \qquad \frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \mathtt{snd}\ e : \tau_2}$$

# Product Types: Dynamic Semantics and Soundness

- New form of values: v ::= ... | $(v_1, v_2)$
- New (big step) evaluation rules:

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2}{(e_1, e_2) \Downarrow (v_1, v_2)}$$

$$\frac{e \Downarrow (v_1, v_2)}{\texttt{fst}\ e \Downarrow v_1} \qquad \frac{e \Downarrow (v_1, v_2)}{\texttt{snd}\ e \Downarrow v_2}$$

- New contexts: H ::= ... | $(H_1, e_2)$ | $(v_1, H_2)$ | fst H | snd H
- New redexes:

  fst $(v_1, v_2) \to v_1$

  snd $(v_1, v_2) \to v_2$

- Type soundness holds just as before

# Records

- Records are like tuples with labels
- New form of expressions

$$e ::= \ldots \mid \{L_1 = e_1, \ldots, L_n = e_n\} \mid e.L$$

- New form of values

$$v ::= \{L_1 = v_1, \ldots, L_n = v_n\}$$

- New form of types

$$\tau ::= \ldots \mid \{L_1 : \tau_1, \ldots, L_n : \tau_n\}$$

- ... follows the model of $F_1^{\times}$
  - typing rules
  - derivation rules
  - type soundness

## Sum Types

- We need types of the form
  - either an int or a float
  - either 0 or a pointer
  - either true or false
  - These are called disjoint union types
- New form of expressions and types

  e ::= … | injl e | injr e |

  case e of injl x → $e_1$ | injr y → $e_2$

  $\tau$ ::= … | $\tau_1 + \tau_2$
  - A value of type $\tau_1 + \tau_2$ is either a $\tau_1$ or a $\tau_2$
  - Like union in C or Pascal, but safe
    - distinguishing between components is under compiler control
  - case is a binding operator: x is bound in $e_1$ and y is bound in $e_2$

## Examples with Sum Types

- Consider the type "unit" with a single element called *
- The type "optional integer" defined as "unit + int"
  - Useful for optional arguments or return values
    - No argument: injl *
    - Argument is 5: injr 5
  - To use the argument you <u>must</u> test the kind of argument
  - case arg of injl x ⇒ "no_arg_case" | injr y ⇒ "...y..."
  - injl and injr are tags and case is tag checking
- Bool is a union type: bool = unit + unit
  - true is injl *
  - false is injr *
  - if e then $e_1$ else $e_2$ is case e of injl x ⇒ $e_1$ | injr y ⇒ $e_2$
  - Check the equivalence of the static and dynamic semantics

# Static Semantics of Sum Types

- New typing rules

$$\frac{\Gamma \vdash e : \tau_1}{\Gamma \vdash \texttt{injl}\ e : \tau_1 + \tau_2} \qquad \frac{\Gamma \vdash e : \tau_2}{\Gamma \vdash \texttt{injr}\ e : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 + \tau_2 \qquad \Gamma, x : \tau_1 \vdash e_l : \tau \qquad \Gamma, y : \tau_2 \vdash e_r : \tau}{\Gamma \vdash \texttt{case}\ e_1\ \texttt{of}\ \texttt{injl}\ x \Rightarrow e_l \mid \texttt{injr}\ y \Rightarrow e_r : \tau}$$

- Types are not unique anymore

  injl 1 : int + bool

  injl 1 : int + (int $\to$ int)

  - this complicates type checking, but still doable

# Dynamic Semantics of Sum Types

- New values v ::= ... | injl v | injr v
- New evaluation rules

$$\frac{e \Downarrow v}{\texttt{injl}\ e \Downarrow \texttt{injl}\ v} \qquad \frac{e \Downarrow v}{\texttt{injr}\ e \Downarrow \texttt{injr}\ v}$$

$$\frac{e \Downarrow \texttt{injl}\ v \qquad [v/x]e_l \Downarrow v'}{\texttt{case}\ e\ \texttt{of}\ \texttt{injl}\ x \Rightarrow e_l \mid \texttt{injr}\ y \Rightarrow e_r \Downarrow v'}$$

$$\frac{e \Downarrow \texttt{injr}\ v \qquad [v/y]e_r \Downarrow v'}{\texttt{case}\ e\ \texttt{of}\ \texttt{injl}\ x \Rightarrow e_l \mid \texttt{injr}\ y \Rightarrow e_r \Downarrow v'}$$

## Type Soundness for $F_1^+$

- Type soundness still holds
- No way to use a $\tau_1 + \tau_2$ inappropriately
- The key is that the only way to use a $\tau_1 + \tau_2$ is with case, which ensures that you are not using a $\tau_1$ as a $\tau_2$
- In C or Pascal checking the tag is the responsibility of the programmer!
  - Unsafe

## Types for Imperative Features

- We looked at types for pure functional languages
- Now we look at types for imperative features
- Such types are used to characterize non-local effects
  - assignments
  - exceptions
- Contextual semantics is useful here

# Reference Types

- Such types are used for mutable memory cells

- Syntax (as in ML)

$$e ::= \ldots \mid \text{ref } e : \tau \mid e_1 := e_2 \mid !\, e$$

$$\tau ::= \ldots \mid \tau \text{ ref}$$

  - ref e - evaluates e, allocates a new memory cell, stores the value of e in it and returns the address of the memory cell
    - like malloc + initialization in C, or new in C++ and Java
  - $e_1 := e_2$, evaluates $e_1$ to a memory cell and updates its value with the value of $e_2$
  - ! e - evaluates e to a memory cell and returns its contents

## Global Effects with Reference Cells

- A reference cell can escape the static scope where it was created

  $(\lambda f{:}int \rightarrow int\ ref.\ !(f\ 5))\quad (\lambda x{:}int.\ ref\ x : int)$

- The value stored in a reference cell must be visible from the entire program

- The "result" of an expression must now include the changes to the heap that it makes

- To model reference cells we must extend the evaluation model

## Modeling References

- A heap is a mapping from addresses to values

  $$h ::= \cdot \mid h, a \leftarrow v : \tau$$

  - $a \in$ Addresses
  - We tag the heap cells with their types
  - Types are useful only for static semantics. They are not needed for the evaluation $\Rightarrow$ not a part of the implementation
- We call a "program" an expression along with a heap

  $$p ::= \text{heap } h \text{ in } e$$

  - The initial program is "heap $\emptyset$ in e"
  - Heap addresses act as bound variables in the expression
  - This is a trick that allows easy reuse of properties of local variables for heap addresses
    - e.g., we can rename the address and its occurrences at will

## Static Semantics of References

- Typing rules for expressions:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash (\texttt{ref}\ e : \tau) : \tau\ \texttt{ref}} \qquad \frac{\Gamma \vdash e : \tau\ \texttt{ref}}{\Gamma \vdash !e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau\ \texttt{ref} \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 := e_2 : \texttt{unit}}$$

- and for programs

$$\frac{\Gamma \vdash v_i : \tau_i\ (i = 1 \ldots n) \quad \Gamma \vdash e : \tau}{\vdash \texttt{heap}\ h\ \texttt{in}\ e : \tau}$$

where $\Gamma = a_1 : \tau_1\ \texttt{ref}, \ldots, a_n : \tau_n\ \texttt{ref}$

and $h = a_1 \leftarrow v_1 : \tau_1, \ldots, a_n \leftarrow v_n : \tau_n$

# Exceptions

- A mechanism that allows non-local control flow
  - Useful for implementing the propagation of errors to caller
- Exceptions ensure that errors are not ignored
  - Compare with the manual error handling in C
- Languages with exceptions:
  - C++, ML, Modula-3, Java

- We assume that there is a special type exn of exceptions
  - exn could be int to model error codes
  - In Java or C++, exn is a special object type

# Modeling Exceptions

- Syntax

  $e ::= \ldots \mid \text{raise } e \mid \text{try } e_1 \text{ handle } x \Rightarrow e_2$

  $\tau ::= \ldots \mid \text{exn}$

- We ignore here how exception values are created
  - In examples we will use integers as exception values
- The handler binds $x$ in $e_2$ to the actual exception value
- The "raise" expression never returns to the immediately enclosing context
  - 1 + raise 2 is well-typed
  - if (raise 2) then 1 else 2 is also well-typed
  - (raise 2) 5 is also well-typed
  - What should the type of raise be?

## Example with Exceptions

- A (strange) factorial function

```
let f = λx:int.λres:int. if x = 0 then
                                   raise res
                              else
                                  f (x - 1) (res * x)
in  try f 5 1 handle x ⇒ x
```

- The function returns in one step from the recursion
- The top-level handler catches the exception and turns it into a regular result

# Typing Exceptions

- New typing rules

$$\frac{\Gamma \vdash e : \texttt{exn}}{\Gamma \vdash \texttt{raise}\ e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, x : \texttt{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{handle}\ x \Longrightarrow e_2 : \tau}$$

- A raise expression has an arbitrary type
  - This is a clear sign that the expression does not return to its evaluation context
- The type of the body of try and of the handler must match
  - Just like for conditionals

# Recursive Types
# Subtyping

# Recursive Types

- It is useful to be able to define recursive data structures
- Example: lists
  - A list of elements of type $\tau$ (a $\tau$ list) is either empty or it is a pair of a $\tau$ and a $\tau$ list

    $$\tau \text{ list} = \text{unit} + (\tau \times \tau \text{ list})$$

  - This is a recursive equation. We take its solution to be the smallest set of values L that satisfies the equation

    $$L = \{*\} \cup (T \times L)$$

    where T is the set of values of type $\tau$
  - Note: this interpretation can be troublesome
    - E.g. $\tau = \tau \to \tau$, but only for trivial sets we have $T = T \to T$
  - Another interpretation is that the recursive equation is up-to set isomorphism

## Recursive Types

- We introduce a recursive type constructor

  $$\mu t.\ \tau$$

  - The type variable t is bound in $\tau$
  - This is the solution to the equation

    $t \simeq \tau$ (t is isomorphic with $\tau$)
  - E.g., $\tau$ list = $\mu t.\ (\text{unit} + \tau \times t)$
  - This allows "unnamed" recursive types
- We introduce syntactic operations for the conversion between $\mu t.\tau$ and $[\mu t.\tau/t]\tau$
- E.g. between "$\tau$ list" and "unit + $\tau \times \tau$ list"

  $e ::= \ldots \mid \text{fold}_{\mu t.\tau}\ e \mid \text{unfold}_{\mu t.\tau}\ e$

  $\tau ::= \ldots \mid t \mid \mu t.\tau$

# Example with Recursive Types

- Lists

  $\tau$ list = $\mu$t. (unit + $\tau \times$ t)

  $nil_\tau$ = $fold_{\tau\ list}$ (injl *)

  $cons_\tau$ = $\lambda$x:$\tau$.$\lambda$L:$\tau$ list. $fold_{\tau\ list}$ injr (x, L)

- A list length function

  $length_\tau$ = $\lambda$L:$\tau$ list. case ($unfold_{\tau\ list}$ L) of injl x $\Rightarrow$ 0
  | injr y $\Rightarrow$ 1 + $length_\tau$ (snd y)

- Verify that
  - $nil_\tau$ : $\tau$ list
  - $cons_\tau$ : $\tau \rightarrow \tau$ list $\rightarrow \tau$ list
  - $length_\tau$ : $\tau$ list $\rightarrow$ int

# Static Semantics of Recursive Types

$$\frac{\Gamma \vdash e : \mu t.\tau}{\Gamma \vdash \mathtt{unfold}_{\mu t.\tau}\ e : [\mu t.\tau/t]\tau}$$

$$\frac{\Gamma \vdash e : [\mu t.\tau/t]\tau}{\Gamma \vdash \mathtt{fold}_{\mu t.\tau}\ e : \mu t.\tau}$$

- The typing rules are syntax directed
- Often, for syntactic simplicity, the fold and unfold operators are omitted
  - This makes type checking somewhat harder

# Dynamics of Recursive Types

- We add a new form of values

  $v ::= \ldots \mid \mathtt{fold}_{\mu t.\tau}\ v$

  - The purpose of fold is to ensure that the value has the recursive type and not its unfolding
- The evaluation rules:

$$\frac{e \Downarrow v}{\mathtt{fold}_{\mu t.\tau}\ e \Downarrow \mathtt{fold}_{\mu t.\tau}\ v} \qquad \frac{e \Downarrow \mathtt{fold}_{\mu t.\tau}\ v}{\mathtt{unfold}_{\mu t.\tau}\ e \Downarrow v}$$

- The folding annotations are for type checking only
- They can be dropped after type checking

## Recursive Types in ML

- The language ML uses a simple syntactic trick to avoid having to write the explicit fold and unfold
- In ML recursive types are bundled with union types

  datatype t = $C_1$ of $\tau_1$ | $C_2$ of $\tau_2$ | ... | $C_n$ of $\tau_n$ (t can appear in $\tau_i$)

  - E.g., datatype intlist = Nil of unit | Cons of int $\times$ intlist
- When the programmer writes

  Cons (5, l)

  - the compiler treats it as

    $\text{fold}_{\text{intlist}}$ (injr (5, l))
- When the programmer writes
  - case e of Nil $\Rightarrow$ ... | Cons (h, t) $\Rightarrow$ ...

  the compiler treats it as

  - case $\text{unfold}_{\text{intlist}}$ e of Nil $\Rightarrow$ ... | Cons (h,t) $\Rightarrow$ ...

# Encoding Call-by-Value λ-calculus in $F_1^\mu$

- So far, $F_1$ was so weak that we could not encode non-terminating computations
  - Cannot encode recursion
  - Cannot write the λx.x x (self-application)
- The addition of recursive types makes typed λ-calculus as expressive as untyped λ-calculus !
- We can show a conversion algorithm from call-by-value untyped λ-calculus to call-by-value $F_1^\mu$

# Untyped Programming in $F_1^{\mu}$

- We write $\underline{e}$ for the conversion of the term e to $F_1^{\mu}$
  - The type of $\underline{e}$ is $V = \mu t.\ t \to t$
- The conversion rules

  $\underline{x} = x$

  $\underline{\lambda x.\ e} = \text{fold}_V\ (\lambda x{:}V.\ \underline{e})$

  $\underline{e_1\ e_2} = (\text{unfold}_V\ \underline{e_1})\ \underline{e_2}$
- Verify that
  1. $\cdot \vdash \underline{e} : V$
  2. $e \Downarrow v$ if and only if $\underline{e} \Downarrow \underline{v}$
- We can express non-terminating computation

  $D = (\text{unfold}_V\ (\text{fold}_V\ (\lambda x{:}V.\ (\text{unfold}_V\ x)\ x)))\ (\text{fold}_V\ (\lambda x{:}V.\ (\text{unfold}_V\ x)\ x)))$

  or, equivalently

  $D = (\lambda x{:}V.\ (\text{unfold}_V\ x)\ x)\ (\text{fold}_V\ (\lambda x{:}V.\ (\text{unfold}_V\ x)\ x)))$

# Subtyping

## Introduction to Subtyping

- Viewing types as denoting sets of values, it is natural to consider a subtyping relation between types as induced by the subset relation between sets
- Informal intuition:
  - If $\tau$ is a subtype of $\sigma$ then any expression with type $\tau$ also has type $\sigma$
  - If $\tau$ is a subtype of $\sigma$ then any expression of type $\tau$ can be used in a context that expects a $\sigma$
  - Subtyping is reflexive and transitive
  - We write $\tau < \sigma$ to say that $\tau$ is a subtype of $\sigma$

## Subtyping Examples

- FORTRAN introduced int < real
  - 5 + 1.5 is well-typed in many languages
- PASCAL had [1..10] < [0..15] < int
- It is generally accepted that subtyping is a fundamental property of object-oriented languages
  - Let S be a subclass of C. Then an instance of S can be used where an instance of C is expected
  - This is "subclassing ⇒ subtyping" philosophy

# Subsumption

- We formalize the informal requirement on subtyping
- Rule of <u>subsumption</u>
  - If $\tau < \sigma$ then an expression of type $\tau$ also has type $\sigma$

$$\frac{\Gamma \vdash e : \tau \quad \tau < \sigma}{\Gamma \vdash e : \sigma}$$

- But now type safety is in danger:
  - If we say that int < int $\rightarrow$ int
  - Then we can prove that "5 5" is well typed !
- There is a way to construct the subtyping relation to preserve type safety

## Defining Subtyping

- The formal definition of subtyping is by derivation rules for the judgment $\tau < \sigma$
- We start with subtyping on the base types
  - E.g. int < real or nat < int
  - These rules are language dependent and are typically based directly on types-as-sets arguments
- We then make subtyping a preorder (reflexive and transitive)

$$\frac{}{\tau < \tau} \qquad \frac{\tau_1 < \tau_2 \quad \tau_2 < \tau_3}{\tau_1 < \tau_3}$$

- Then we build-up subtyping for "larger" types

# Subtyping for Pairs

- Try

$$\frac{\tau < \sigma \quad \tau' < \sigma'}{\tau \times \tau' < \sigma \times \sigma'}$$

- Show (informally) that whenever a $\sigma \times \sigma'$ can be used, a $\tau \times \tau'$ can also be used:
- Consider the context H = H' [fst •] expecting a $\sigma \times \sigma'$
  - Then H' expects a $\sigma$
  - Because $\tau < \sigma$ then H' accepts a $\tau$
  - Take e : $\tau \times \tau'$. Then fst e : $\tau$ so it works in H'
  - Thus e works in H
- The case of "snd •" is similar

## Subtyping for Functions

- Try the (naive) rule

$$\frac{\tau < \sigma \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$

- This rule is unsound
  - Let $\Gamma$ = f : int $\to$ bool   (and assume int < real)
  - We show using the above rule that $\Gamma \vdash$ f  5.0 : bool
  - But this is wrong since 5.0 is not a valid argument of f

$$\frac{\dfrac{\Gamma \vdash f : \texttt{int} \to \texttt{bool} \qquad \dfrac{\texttt{int} < \texttt{real} \quad \texttt{bool} < \texttt{bool}}{\texttt{int} \to \texttt{bool} < \texttt{real} \to \texttt{bool}}}{\Gamma \vdash f : \texttt{real} \to \texttt{bool}} \qquad \Gamma \vdash 5.0 : \texttt{real}}{\Gamma \vdash f\ 5.0 : \texttt{bool}}$$

# Subtyping for Functions (Cont.)

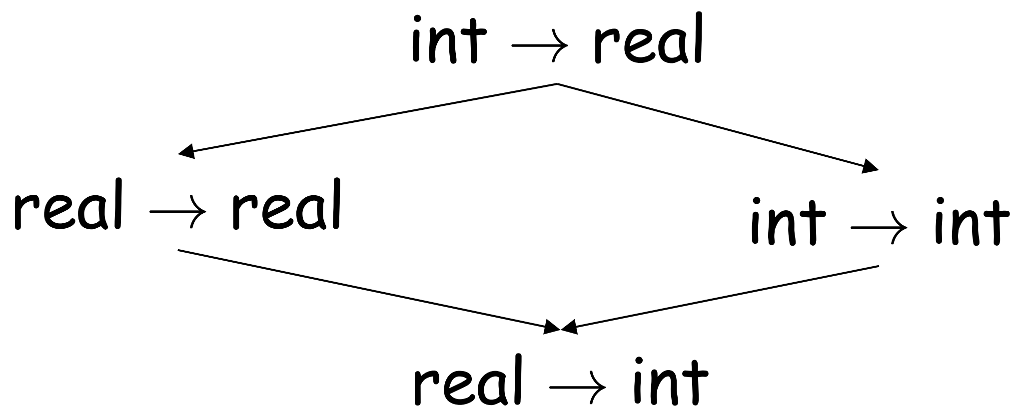
- The correct rule

$$\frac{\sigma < \tau \quad \tau' < \sigma'}{\tau \to \tau' < \sigma \to \sigma'}$$

- We say that $\to$ is covariant in the result type and contravariant in the argument type
- Informal correctness argument:
  - Pick f : $\tau \to \tau'$
  - f expects an argument of type $\tau$
  - It also accepts an argument of type $\sigma < \tau$
  - f returns a value of type $\tau'$
  - Which can also be viewed as a $\sigma'$ (since $\tau' < \sigma'$)
  - Hence f can be used as $\sigma \to \sigma'$

## More on Contravariance

- Consider the subtype relationships




- In what sense $f \in$ real $\to$ int $\Rightarrow f \in$ int $\to$ int?
  - "real $\to$ int" has a larger domain!

- This suggests that "subtype-as-subset" interpretation is not straightforward

# Subtyping References

- Try covariance

$$\frac{\tau < \sigma}{\tau \ \mathtt{ref} < \sigma \ \mathtt{ref}}$$

Wrong!

  - Example: assume $\tau < \sigma$
  - The following holds (if we assume the above rule):

$$x : \sigma, y : \tau \ \text{ref}, f : \tau \to \text{int} \vdash y := x; f\ (!\ y)$$

  - Unsound: f is called on a $\sigma$ but is defined only on $\tau$
  - Java has covariant arrays !
- If we want covariance of references we can recover type safety with a runtime check for each y := x
  - The actual type of x matches the actual type of y
  - But this is generally considered a bad design

# Subtyping References (Cont.)

- Try contravariance:

$$\frac{\tau < \sigma}{\sigma\ \mathtt{ref} < \tau\ \mathtt{ref}}$$

Also Wrong!

  - Example: assume $\tau < \sigma$
  - The following holds (if we assume the above rule):

    $$x : \sigma, y : \sigma\ \text{ref}, f : \tau \to \text{int} \vdash y := x;\ f\ (!\ y)$$

  - Unsound: f is called on a $\sigma$ but is defined only on $\tau$
- References are <u>invariant</u>
  - no subtyping for references (unless we are prepared to add run-time checks)
  - hence, arrays should be invariant
  - hence, mutable records should be invariant

## Subtyping Recursive Types

- Recall $\tau$ list = $\mu$t.(unit + $\tau \times$t)
  - We would like $\tau$ list < $\sigma$ list whenever $\tau < \sigma$
- Try simple covariance:

$$\frac{\tau < \sigma}{\mu t.\tau < \mu t.\sigma} \qquad \text{Wrong!}$$

- This is wrong if t occurs contravariantly in $\tau$
- Take $\tau = \mu t.t \rightarrow int$ and $\sigma = \mu t.t \rightarrow real$
- Above rule says that $\tau < \sigma$
- We have $\tau \simeq \tau \rightarrow int$ and $\sigma \simeq \sigma \rightarrow real$
- $\tau < \sigma$ would mean covariant function type!
- How can we still have the subtyping for lists?

# Subtyping Recursive Types (Cont.)

- The correct rule

$$\frac{\begin{array}{c} t < s \\ \vdots \\ \tau < \sigma \end{array}}{\mu t.\tau < \mu s.\sigma}$$

- We add as an assumption that the type variables stand for types with the desired subtype relationship
  - Before we assumed that they stand for the <u>same</u> type!

- Verify that subtyping now works properly for lists

- There is no subtyping between $\mu t.t \to \text{int}$ and $\mu t.t \to \text{real}$

# Second-Order Type Systems

## The Limitations of $F_1$

- In $F_1$ each function works exactly for one type
- Example: sorting function
  - sort : $(\tau \to \tau \to \text{bool}) \to \tau \text{ array} \to \text{unit}$
- The various sorting functions differ only in typing
  - At runtime they perform exactly the same operations
  - Need different versions only to keep the type checker happy
- Two alternatives:
  - Circumvent the type system (example: C, Java), or
  - Use a more flexible type system that lets us write only one sorting function (example: ML, Java 1.5)

## Polymorphism

- Informal definition

  A function is polymorphic if it can be applied to "many" types of arguments
- Various kinds of polymorphism depending on the definition of "many"
  - subtype (or bounded) polymorphism
    - "many" = all subtypes of a given type
  - ad-hoc polymorphism
    - "many" = depends on the function
    - choose behavior at runtime (depending on types, e.g. sizeof)
  - parametric predicative polymorphism
    - "many" = all monomorphic types
  - parametric impredicative polymorphism
    - "many" = all types

# Parametric Polymorphism: Types as Parameters

- We introduce type variables and allow expressions to have variable types
- We introduce polymorphic types

$$\tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t \mid \forall t.\ \tau$$

$$e ::= x \mid \lambda x{:}\tau.e \mid e_1\ e_2 \mid \Lambda t.\ e \mid e[\tau]$$

  - $\Lambda t.\ e$ is type abstraction (or *generalization*)
  - $e[\tau]$ is type application (or instantiation)
- Examples:
  - $id = \Lambda t.\lambda x{:}t.\ x \quad : \quad \forall t.t \rightarrow t$
  - $id[int] = \lambda x{:}int.\ x \quad : \quad int \rightarrow int$
  - $id[bool] = \lambda x{:}bool.\ x \quad : \quad bool \rightarrow bool$
  - "id 5" is invalid. Use "id [int] 5" instead

# Impredicative Polymorphism

- The typing rules:

$$\frac{x : \tau \text{ in } \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x : \tau . e : \tau \to \tau'}$$

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1\ e_2 : \tau'}$$

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash \Lambda t . e : \forall t . \tau} \quad t \text{ does not occur in } \Gamma$$

$$\frac{\Gamma \vdash e : \forall t . \tau'}{\Gamma \vdash e[\tau] : [\tau / t]\tau'}$$

## Impredicative Polymorphism (Cont.)

- Verify that "id [int] 5" has type int
- Note the side-condition in the rule for type abstraction
  - Prevents ill-formed terms like: $\lambda x{:}t.\Lambda t.x$
- The evaluation rules are just like those of $F_1$
  - This means that type abstraction and application are all performed at compile time
  - We do not evaluate under $\Lambda$ ($\Lambda t.\ e$ is a value)
  - We do not have to operate on types at run-time
  - This is called phase separation: type checking and execution

# Expressiveness of Impredicative Polymorphism

- This calculus is called
  - $F_2$
  - system F
  - second-order $\lambda$-calculus
  - polymorphic $\lambda$-calculus

- Polymorphism is extremely expressive

- We can encode many base and structured types in $F_2$

## What's Wrong with $F_2$

- Simple syntax but very complicated semantics
  - id can be applied to itself: "id [$\forall$t. t $\to$ t] id"
  - This can lead to paradoxical situations in a pure set-theoretic interpretation of types
  - E.g., the meaning of id is a function whose domain contains a set (the meaning of $\forall$t.t$\to$ t) that contains id !
  - This suggests that giving an interpretation to impredicative type abstraction is tricky
- Complicated termination proof (Girard)
- Type reconstruction (typeability) is undecidable
  - If the type application and abstraction are missing
- How to fix it?
  - Restrict the use of polymorphism

## Predicative Polymorphism

- Restriction: type variables can be instantiated only with monomorphic types
- This restriction can be expressed syntactically

  $\tau ::= b \mid \tau_1 \to \tau_2 \mid t$

  $\sigma ::= \tau \mid \forall t.\ \sigma \mid \sigma_1 \to \sigma_2$

  $e ::= x \mid e_1\ e_2 \mid \lambda x{:}\sigma.\ e \mid \Lambda t.e \mid e\ [\tau]$

  - Type application is restricted to mono types
  - Cannot apply “id” to itself anymore

- Same typing rules
- Simple semantics and termination proof
- Type reconstruction still undecidable
- Must restrict further !

## Prenex Predicative Polymorphism

- Restriction: polymorphic type constructor at top level only
- This restriction can also be expressed syntactically

  $\tau ::= b \mid \tau_1 \to \tau_2 \mid t$

  $\sigma ::= \tau \mid \forall t.\ \sigma$

  $e ::= x \mid e_1\ e_2 \mid \lambda x{:}\tau.\ e \mid \Lambda t.e \mid e\ [\tau]$

  - Type application is restricted to mono types (i.e., predicative)
  - Abstraction only on mono types
  - The only occurrences of $\forall$ are at the top level of a type

    $(\forall t.\ t \to t) \to (\forall t.\ t \to t)$ is <u>not</u> a valid type
- Same typing rules
- Simple semantics and termination proof
- Decidable type inference !

# Expressiveness of Prenex Predicative $F_2$

- We have simplified too much !
- Not expressive enough to encode nat, bool
  - But such encodings are only of theoretical interest anyway
- Is it expressive enough in practice?
  - Almost
  - Cannot write something like

  $(\lambda s{:}\forall t.\tau.\ \ldots\ s\ [nat]\ x\ \ldots\ \ s\ [bool]\ y)\ (\Lambda t.\ \ldots$ code for sort$)$
  - Because the type of formal argument s cannot be polymorphic

# ML's Polymorphic Let

- ML solution: slight extension of the predicative $F_2$
  - Introduce "let x : $\sigma$ = $e_1$ in $e_2$"
  - With the semantics of "($\lambda$x : $\sigma$.$e_2$) $e_1$"
  - And typed as "[$e_1$/x] $e_2$"

$$\frac{\Gamma \vdash e_1 : \sigma \qquad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x : \sigma\ = e_1\ \texttt{in}\ e_2 : \tau}$$

- This lets us write the polymorphic sort as

  let

  s : $\forall$t.$\tau$ = $\Lambda$t. ... code for polymorphic sort ...

  in

  ... s [nat] x .... s [bool] y
- Surprise: this was a major ML design flaw!

## ML Polymorphism and References

- let is evaluated using call-by-value but is typed using call-by-name
  - What if there are side effects ?
- Example:

  let $x : \forall t. (t \to t)$ ref = $\Lambda t$.ref ($\lambda x : t. x$)

  in

    x [bool] := $\lambda x$: bool. not x

    (! x [int]) 5

  end

  - Will apply "not" to 5
  - Similar examples can be constructed with exceptions
- It took 10 years to find and agree on a clean solution

# The Value Restriction in ML

- *A* type in a let is generalized only for syntactic values

$$\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \texttt{let}\ x : \sigma = e_1\ \texttt{in}\ e_2 : \tau}$$

$e_1$ is a syntactic value or $\sigma$ is monomorphic

- Since $e_1$ is a value, its evaluation cannot have side-effects
- In this case call-by-name and call-by-value are the same
- In the previous example ref ($\lambda$x:t. x) is not a value
- This is not too restrictive in practice !

## Subtype Bounded Polymorphism

- We can bound the instances of a given type variable

  $\forall t < \tau.\ \sigma$

- Consider a function $f : \forall t < \tau.\ t \to \sigma$
- How is this different from $f' : \tau \to \sigma$
  - We can also invoke $f'$ on any subtype of $\tau$

- They are different if $t$ appears in $\sigma$
  - E.g, $f : \forall t{<}\tau.t \to t$ and $f' : \tau \to \tau$
  - Take $x : \tau' < \tau$
  - We have $f\ [\tau']\ x : \tau'$
  - And $f'\ x : \tau$
  - We lost information with $f'$

## Not covered in this class

- A lot!
- Dependent Types
- Types for abstraction and modularity
- Pi calculus
- Object calculi
- Type-based analysis
- Constraint-based analysis
- Applications (looked at some)
- And more …