

Second-Order Type Systems

Lecture 19 CS263

CS 263 Lecture 19

1

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

CS 263 Lecture 19

3

Impredicative Polymorphism

- The typing rules:

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

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

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

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

CS 263 Lecture 19

5

The Limitations of F_1

- In F_1 each function works exactly for one type
- Example: the identity function
 - $\text{id} = \lambda x : \tau. x : \tau \rightarrow \tau$
 - We need to write one version for each type
 - Even more important: $\text{sort} : (\tau \rightarrow \tau \rightarrow \text{bool}) \rightarrow \tau \text{ array} \rightarrow \text{unit}$
- The various sorting functions differ only in typing
 - At runtime they perform exactly the same operations
 - We need different versions only to keep the type checker happy
- Two alternatives:
 - Circumvent the type system (see C, Java, ...), or
 - Use a more flexible type system that lets us write only one sorting function

CS 263 Lecture 19

2

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:
 - $\text{id} = \Lambda t. \lambda x : t. x \quad : \forall t. t \rightarrow t$
 - $\text{id}[\text{int}] = \lambda x : \text{int}. x \quad : \text{int} \rightarrow \text{int}$
 - $\text{id}[\text{bool}] = \lambda x : \text{bool}. x \quad : \text{bool} \rightarrow \text{bool}$
 - "id 5" is invalid. Use "id [int] 5" instead

CS 263 Lecture 19

4

Impredicative Polymorphism (Cont.)

- Verify that "id [int] 5" has type int

- 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 Λ ($\Lambda t. e$ is a value)
 - We do not have to operate on types at run-time
 - This is called phase separation: type checking and execution

CS 263 Lecture 19

6

Parametricity or "Theorems for Free" (P. Wadler)

- Based on the type of a term we can prove properties of that term
- There is only one value of type $\forall t. t \rightarrow t$
 - The identity function
- There is no value of type $\forall t. t$
- Take the function
 $\text{reverse} : \forall t. t \text{ List} \rightarrow t \text{ List}$
 - This function cannot inspect the elements of the list
 - It can only produce a permutation of the original list
 - If L_1 and L_2 have the same length and let "match" be a function that compares two lists element-wise according to an arbitrary predicate
 - then "match $L_1 L_2" \Rightarrow$ "match (reverse L_1) (reverse L_2)" !

CS 263 Lecture 19

7

Expressiveness of Impredicative Polymorphism

- This calculus is called
 - F_2
 - system F
 - second-order λ -calculus
 - polymorphic λ -calculus
- Polymorphism is extremely expressive
- We can encode many base and structured types in F_2

CS 263 Lecture 19

8

Encoding Base Types in F_2

- Booleans
 - $\text{bool} = \forall t. t \rightarrow t \rightarrow t$ (given any two things, select one)
 - There are exactly two values of this type !
 - $\text{true} = \lambda t. \lambda x:t. \lambda y:t. x$
 - $\text{false} = \lambda t. \lambda x:t. \lambda y:t. y$
 - $\text{not} = \lambda b:\text{bool}. \lambda t. \lambda x:t. \lambda y:t. b [t] y x$
- Naturals
 - $\text{nat} = \forall t. (t \rightarrow t) \rightarrow t \rightarrow t$ (given a successor and a zero element, compute a natural number)
 - $0 = \lambda t. \lambda s:t \rightarrow t. \lambda z:t. z$
 - $n = \lambda t. \lambda s:t \rightarrow t. \lambda z:t. s (s (...s(z)))$
 - $\text{add} = \lambda n:\text{nat}. \lambda m:\text{nat}. \lambda t. \lambda s:t \rightarrow t. \lambda z:t. n [t] s (m [t] s z)$
 - $\text{mul} = \lambda n:\text{nat}. \lambda m:\text{nat}. \lambda t. \lambda s:t \rightarrow t. \lambda z:t. n [t] (m [t] s) z$

CS 263 Lecture 19

9

Expressiveness of F_2

- We can encode similarly:
 - $\tau_1 + \tau_2$ as $\forall t. (\tau_1 \rightarrow t) \rightarrow (\tau_2 \rightarrow t) \rightarrow t$
 - $\tau_1 \times \tau_2$ as $\forall t. (\tau_1 \rightarrow \tau_2 \rightarrow t) \rightarrow t$
 - unit as $\forall t. t \rightarrow t$
- We cannot encode $\mu t. \tau$
 - We can encode primitive recursion but not full recursion
 - All terms in F_2 have a termination proof in second-order Peano arithmetic (Girard, 1971)
 - This is the set of naturals defined using zero, successor, induction along with quantification both over naturals and over sets of naturals

CS 263 Lecture 19

10

What's Wrong with F_2

- Simple syntax but very complicated semantics
 - id can be applied to itself: "id $[\forall t. t \rightarrow t]$ 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 $\forall t. t \rightarrow t$) 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

CS 263 Lecture 19

11

Predicative Polymorphism

- Restriction: type variables can be instantiated only with monomorphic types
- This restriction can be expressed syntactically
 - $\tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t$
 - $\sigma ::= \tau \mid \forall t. \sigma \mid \sigma_1 \rightarrow \sigma_2$
 - $e ::= x \mid e_1 e_2 \mid \lambda x:\sigma. e \mid \lambda t.e \mid e [t]$
 - 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 !

CS 263 Lecture 19

12

Prenex Predicative Polymorphism

- Restriction: polymorphic type constructor at top level only
- This restriction can also be expressed syntactically
 - $\tau ::= b \mid \tau_1 \rightarrow \tau_2 \mid t$
 - $\sigma ::= \tau \mid \forall t. \sigma$
 - $e ::= x \mid e_1 e_2 \mid \lambda x:\tau. e \mid \Lambda t.e \mid e [\tau]$
 - 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
 $(\forall t. t \rightarrow t) \rightarrow (\forall t. t \rightarrow t)$ is not a valid type
- Same typing rules
- Simple semantics and termination proof
- Decidable type inference !

CS 263 Lecture 19

13

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

CS 263 Lecture 19

14

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 " $[e_1/x] e_2$ "

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

- This lets us write the polymorphic sort as
 - let
 - $s : \forall t.t = \Lambda t. \dots \text{code for polymorphic sort} \dots$
 - in
 - $\dots s [\text{nat}] x \dots s [\text{bool}] y$
- Surprise: this was a major ML design flaw!

CS 263 Lecture 19

15

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 $x : \forall t. (t \rightarrow t) \text{ ref} = \Lambda t. \text{ref } (\lambda x : t. x)$
 - in
 - $x [\text{bool}] := \lambda x: \text{bool}. \text{not } x$
 - $(! x [\text{int}]) 5$
 - end
 - Will apply "not" to 5
 - Similar examples can be constructed with exceptions
- It took years to find and agree on a clean solution

CS 263 Lecture 19

16

The Value Restriction in ML

- A type in a let is generalized only for syntactic values
 - $\frac{\Gamma \vdash e_1 : \sigma \quad \Gamma, x : \sigma \vdash e_2 : \tau}{\Gamma \vdash \text{let } x : \sigma = e_1 \text{ in } e_2 : \tau}$ e_1 is a syntactic value or σ 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 $\text{ref } (\lambda x:t. x)$ is not a value
- This is not too restrictive in practice !

CS 263 Lecture 19

17

Subtype Bounded Polymorphism

- We can bound the instances of a given type variable
 - $\forall t < \tau. \sigma$
- Consider a function $f : \forall t < \tau. t \rightarrow \sigma$
- How is this different than $f' : \tau \rightarrow \sigma$
 - We can also invoke f' on any subtype of τ
- They are different if t appears in σ
 - E.g, $f : \forall t < \tau. t \rightarrow t$ and $f' : \tau \rightarrow \tau$
 - Take $x : \tau' < \tau$
 - We have $f [\tau'] x : \tau'$
 - And $f' x : \tau$
 - We lost information with f'

CS 263 Lecture 19

18

Types for Data Abstraction

CS 263 Lecture 19

19

Example of Abstraction

- Cartesian points
- Introduce the "abstype" language construct:
 - abstype point implements
 - mk : real × real → point
 - xc : point → real
 - yc : point → real
 - is
 - < point = real × real,
 - mk = λx. x,
 - xc = fst,
 - yc = snd >
- Shows a concrete implementation
- Allows the rest of the program to access the implementation through an abstract interface
- Only the interface need to be publicized
- Allows separate compilation

CS 263 Lecture 19

21

Example with Abstraction

- $C = \{mk = \lambda x.x, xc = fst, yc = snd\}$ is a concrete implementation of points as $real \times real$
- We want to hide the type of the representation σ is the following type:
 - $\{mk : real \times real \rightarrow point, xc : point \rightarrow real, yc : point \rightarrow real\}$
- Note that $C : [real \times real / point] \sigma$
- $A = \langle point = real \times real, C : \sigma \rangle$ is an expression of the abstract type $\exists point. \sigma$
- We want clients to access only the second component of A and just use the abstract name "point" for the first component:
 - open A as point, P : σ in ... P.xc(P.mk(1.0, 2.0)) ...

CS 263 Lecture 19

23

Data Abstraction

- Ability to hide (abstract) concrete implementation details
- Modularity builds on data abstraction
- Improves program structure and minimizes dependencies
- One of the most influential developments of the 1970s
- Key element for much of the success of object orientation in the 1980s

CS 263 Lecture 19

20

Data Abstraction

- It is useful to separate the creation of the abstract type and its use
- Extend the syntax:
 - Terms ::= ... | < t = $\tau, e : \sigma$ > | open e_a as t, x : σ in e_b
 - Types ::= ... | $\exists t. \sigma$
- The expression <t= $\tau, e : \sigma$ > takes the concrete implementation e and "packs it" as a value of an abstract type
 - Alternative notation: "pack e as $\exists t. \sigma$ with $t = \tau$ "
- The "open" expression allows e_b to access the abstract type expression e_a using the name x , the unknown type of the concrete implementation "t" and the interface σ

CS 263 Lecture 19

22

Typing Rules for Existential Types

- We add the following typing rules:

$$\frac{\Gamma \vdash [\tau/t]e : [\tau/t]\sigma}{\Gamma \vdash \langle t = \tau, e : \sigma \rangle : \exists t. \sigma}$$

$$\frac{\Gamma \vdash e_a : \exists t. \sigma \quad \Gamma, t, p : \sigma \vdash e_b : \tau}{\Gamma \vdash \text{open } e_a \text{ as } t, p : \sigma \text{ in } e_b : \tau}$$

- We need the restriction $t \notin FV(\Gamma, \tau)$
 - To ensures that t is bound in e_b and σ

CS 263 Lecture 19

24

Evaluation Rules for Abstract Types

- We add a new form of value

$$v ::= \dots \mid \langle t = \tau, v : \sigma \rangle$$
 - This is just like v but with some type decorations that make it have an existential type

$$\frac{e_a \Downarrow \langle t = \tau, v : \sigma \rangle \quad [v/x][\tau/t]e_b \Downarrow v}{\text{open } e_a \text{ as } t, p : \sigma \text{ in } e_b \Downarrow v}$$
- At the time e_b is evaluated, abstract-type variables are replaced with concrete values
 - If we ignore the type issues "open e_a as $t, x : \sigma$ in e_b " is like "let $x : \sigma = e_a$ in e_b "
 - What is different is that e_b cannot know statically what is the concrete type of x so it cannot take advantage of it

CS 263 Lecture 19

25

Abstract Types as a Specification Mechanism

- Just like polymorphism, existential types are mostly a type checking mechanism
- A function of type $\forall t. t \text{ List} \rightarrow \text{int}$ does not know statically what is the type of the list elements. Therefore no operations are allowed on them
 - But the function will know at run-time the actual value of t
 - "There are no type variables at run-time"
- Same goes for existentials
- These type mechanisms are very powerful form of static checking
 - Recall the "theorems for free"

CS 263 Lecture 19

26

Data Abstraction and Static Checking

- Example: file descriptors
- Solution 1:
 - Represent file descriptors as "int" and export the interface $\{\text{open} : \text{string} \rightarrow \text{int}, \text{read} : \text{int} \rightarrow \text{data}\}$
- An untrusted client of the interface calls "read"
- How can we know that "read" is invoked with a file descriptor that was obtained from "open"?
 - We must keep track of all integers that represent file descriptors
 - We design the interface such that all such integers are small integers and we can essentially keep a bitmap
 - This becomes expensive with more complex (e.g. pointer-based) representations

CS 263 Lecture 19

27

Data Abstraction and Static Checking

- Solution 2:
 - Use the same representation but export an abstraction of it $\exists \text{fd}. \text{File} \text{ or } \exists \text{fd}. \{\text{open} : \text{string} \rightarrow \text{fd}, \text{read} : \text{fd} \rightarrow \text{data}\}$
 - A possible value: $\text{Fd} = \langle \text{fd} = \text{int}, \{\text{open} = \dots, \text{read} = \dots\} : \text{File} \rangle : \exists \text{fd}. \text{File}$
- Now the untrusted client e

$$\text{open Fd as fd, } x : \text{File in } e$$
- At run-time "e" can see that file descriptors are integers
 - But that will not help. It cannot cast 7 as a file descriptor
 - Static checking with no run-time costs!
- Catch: you must be able to type check "e"

CS 263 Lecture 19

28

Modularity

- A module is a program fragment along with visibility constraints
- Visibility of functions and data
 - Specify the function interface but hide its implementation
- Visibility of type definitions
 - More complicated because the type might appear in specifications of the visible functions and data
 - Can use data abstraction to handle this
- A module is represented as a type component and an implementation component

$$\langle t = \tau, e : \sigma \rangle \quad (\text{where } t \text{ can occur in } e \text{ and } \sigma)$$
 - even though the specification (σ) refers to the implementation type we can still hide the latter

CS 263 Lecture 19

29

Problems with Existentials

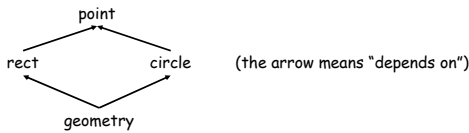
- Existential types
 - Allow representation (type) hiding
 - Allow separate compilation. Need to know only the type of a module to compile its client
 - First-class modules. They can be selected at run-time
- Problems:
 - Closed scope. Must open an existential before it's used
 - Poor support for module hierarchies

CS 263 Lecture 19

30

Problems with Existentials (Cont.)

- There is an inherent tension between handling modules in isolation (good for separate compilation, interchangeability) and the need to integrate them



- Solution 1: open "point" at top level
 - Inversion of program structure
 - The most basic construct has the widest scope

CS 263 Lecture 19

31

Give Up Abstraction?

- Solution 2: incorporate point in rect and circle
 - $R = \langle \text{point} = \dots, \langle \text{rect} = \text{point} \times \text{point}, \dots \rangle \dots \rangle$
 - $C = \langle \text{point} = \dots, \langle \text{circle} = \text{point} \times \text{real}, \dots \rangle \dots \rangle$
- When we open R and C we get two distinct notions of point!
 - And we will not be able to combine them
- Another option is to allow the type checker to see the representation type
 - and thus give up representation hiding

CS 263 Lecture 19

32

Strong Sums

- New way to open a package
 - Terms $e ::= \dots \mid \text{Ops}(e)$
 - Types $\tau ::= \dots \mid \Sigma t. \tau \mid \text{Typ}(e)$
 - Use Typ and Ops to decompose the module
 - Operationally, they are just like the usual "fst" and "snd"
 - $\Sigma t. \tau$ is the dependent sum type
 - It is like $\exists t. \tau$ except we can look at the type

$$\frac{\Gamma \vdash e : \Sigma t. \tau}{\Gamma \vdash \text{Ops}(e) : \tau[\text{Typ}(e)/t]}$$

CS 263 Lecture 19

33

Modularity with Strong Sums

- Example:
 - Consider the R and C defined as before:
 - $\text{Pt} = \langle \text{point} = \text{real} \times \text{real}, \dots \rangle : \Sigma \text{point}. \tau_p$
 - $R = \langle \text{point} = \text{Typ}(\text{Pt}), \langle \text{rect} = \text{point} \times \text{point}, \dots \rangle : \Sigma \text{rect}. \tau_R$
 - $C = \langle \text{point} = \text{Typ}(\text{Pt}), \langle \text{circle} = \text{point} \times \text{real}, \dots \rangle : \Sigma \text{circle}. \tau_C$
- Since we use strong-sums the type checker sees that the two point types are the same

CS 263 Lecture 19

34

Modules with Strong Sums

- ML's module is based on strong sums
- Problems:
 - Poor data abstraction
 - Expressions appear in types (Typ(e))
 - Types might not be known until at run time
 - Lost separate compilation
 - Trouble if e has side-effects (but we can use a value restriction)
 - Second-class modules (because of value restriction)
 - We can combine existentials with strong sums
 - Translucent sums: partially visible

CS 263 Lecture 19

35