Functional Languages

Functional Languages (Applicative, valueoriented)

Importance?

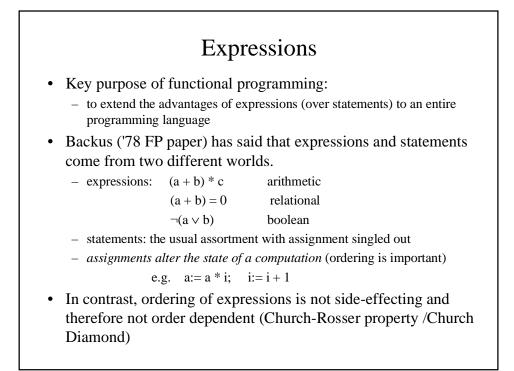
- In their pure form they dispense with notion of assignment
 - claim is: it's easier to program in them

- independence of evaluation order

- also: easier to reason about programs written in them
- FPL's encourage thinking at higher levels of abstraction
 - support modifying and combining existing programs
 - thus, FPL's encourage programmers to work in units larger than statements of conventional languages: "programming in the large"
- FPL's provide a paradigm for parallel computing
 - absence of assignment (single assignment) } provide basis
 - } for parallel
 - ability to operate on entire data structures } functional programming

Importance of Functional Languages...

- Valuable in developing <u>executable specifications</u> and prototype implementations
 - Simple underlying semantics
 - rigorous mathematical foundations
 - ability to operate on entire data structures
 - => ideal vehicle for capturing specifications
- Utility to AI
 - Most AI done in func langs (extensibility. symbolic manipulation)
- Functional Programming is tied to CS theory
 - provides framework for viewing decidability questions
 - (both programming and computers)
 - Good introduction to Denotational Semantics
 - functional in form



More Expressions

- With Church-Rosser
 - reasoning about expressions is easier
 - order independence supports fine-grained parallelism
 - Diamond property is quite useful
- Referential transparency
 - In a fixed context, the replacement of a subexpression by its value is completely independent of the surrounding expression
 - having once evaluated an expression in a given context, shouldn't have to do it again.

Alternative: referential transparency is the universal ability to substitute equals for equals (useful in common subexpression optimizations and mathematical reasoning)

Hoare's Principles of Structuring

(1973, "Hints on Programming Language Design," Stanford Tech Rep)

- 1) Transparency of meaning
 - Meaning of whole expression can be understood in terms of meanings of its subexpressions.

2) Transparency of Purpose

 Purpose of each part consists solely of its contribution to the purpose of the whole.

3) Independence of Parts

- Meaning of independent parts can be understood completely independently.
 - In E + F, E can be understood independently of F.

Hoare's Principles of Structuring

4) Recursive Application

- Both construction and analysis of structure (e.g. expressions) can be accomplished through recursive application of uniform rules.

5) Narrow Interfaces

- Interface between parts is <u>clear</u>, narrow (minimal number of inputs and outputs) and well controlled.

6) Manifestness of Structure

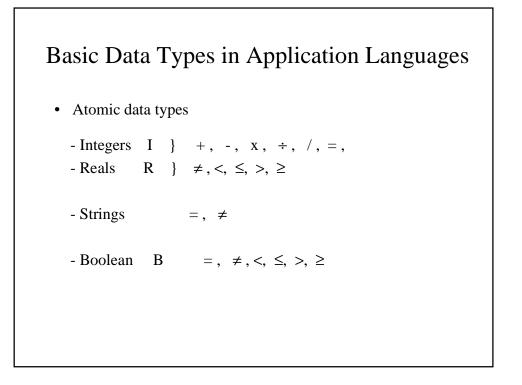
 Structural relationships among parts are obvious. e.g. one expression is subexpression of another if the first is textually embedded in the second. Expressions are unrelated if they are not structurally related.

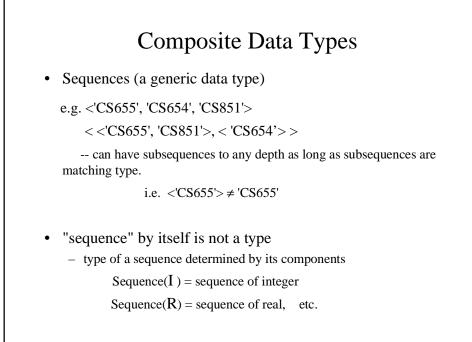
Properties of Pure Expressions

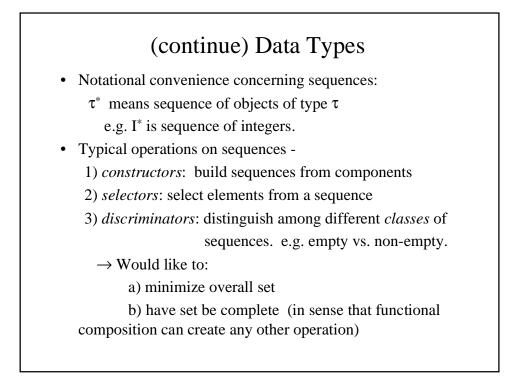
- Value is independent of evaluation order
- Expressions can be evaluated in parallel
- Referential transparency
- No side-effects (Church Rosser)
- Inputs to an expression are obvious from written form
- Effects of operation are obvious from written form
 - \rightarrow Meet Hoare's principles well
 - \rightarrow Good attributes to extend to all programming (?)

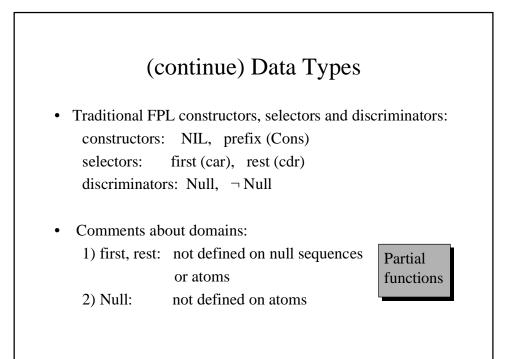
A Scheme Continuation

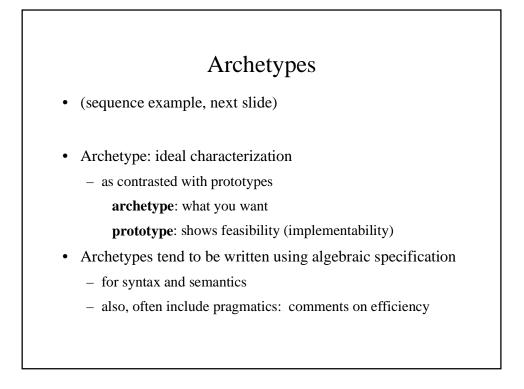
```
(set init-rand (lambda (seed)
  (lambda () (set seed (mod (+ (* seed 9) 5) 1025)))))
(set rand (init-rand 1))
• Sequence of calls to (rand) produces a changing set of values
• So much for referential transparency...
```

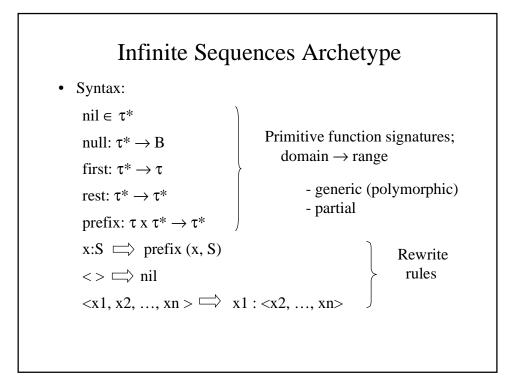


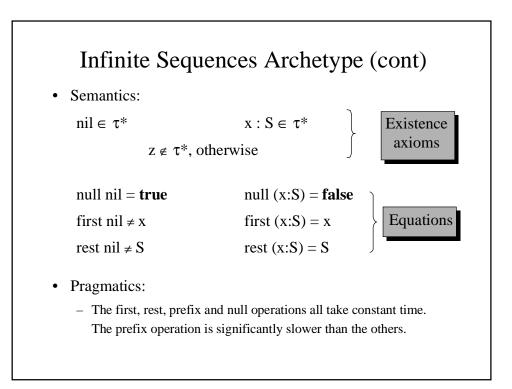












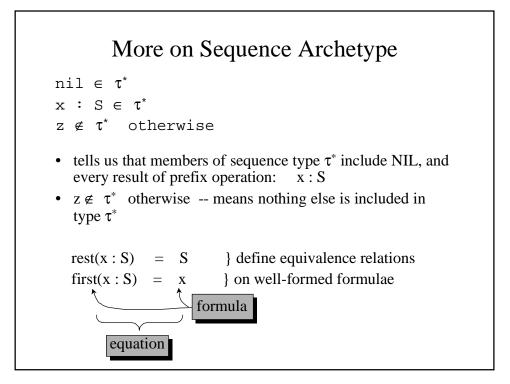
Notes on Sequence Archetype

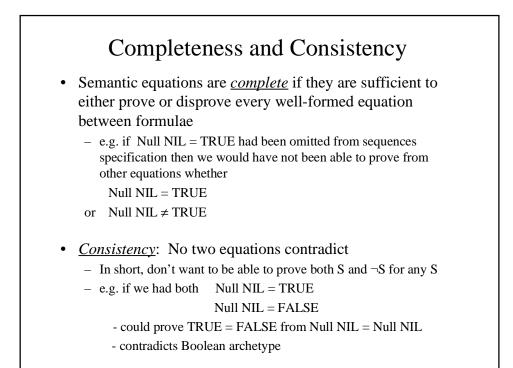
a) $\langle x_1, x_2, \dots x_n \rangle \rightarrow x_1 : \langle x_2, \dots x_n \rangle$ $\rightarrow x_1 : x_2 : x_3 \dots : x_n : NIL$

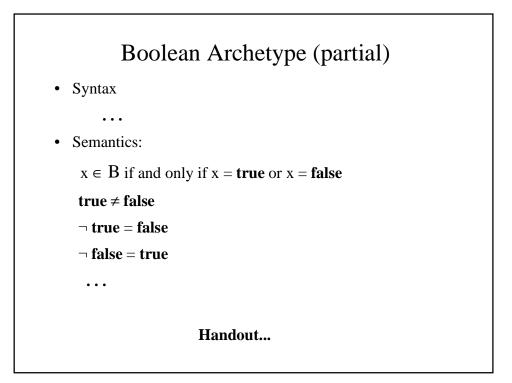
b) domains, ranges and signatures are important concepts

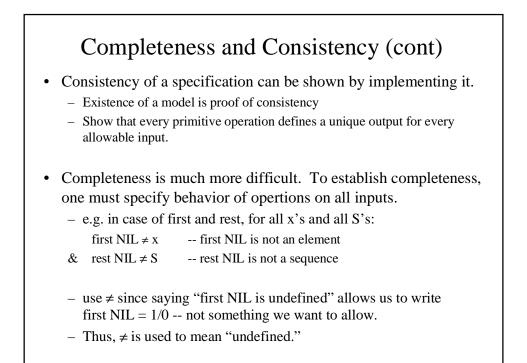
c) defined functions are *generic*

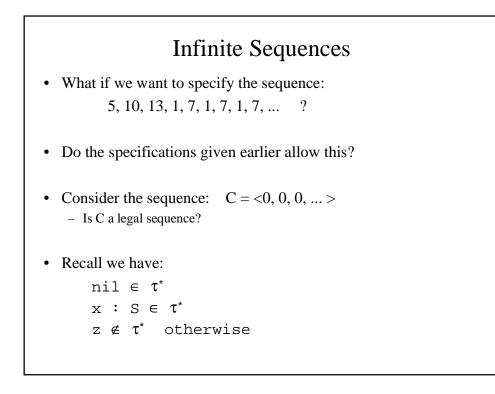
d) defined functions are *partial*First, rest don't work on null sequences

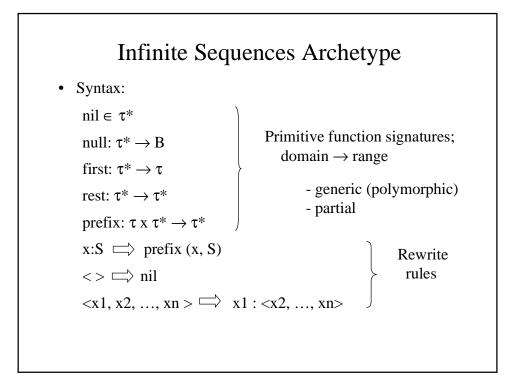


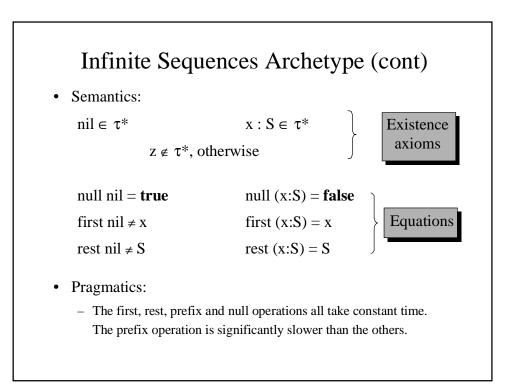


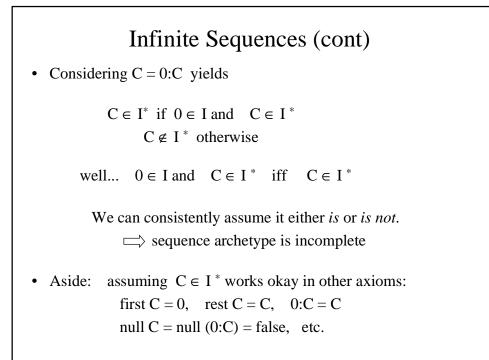


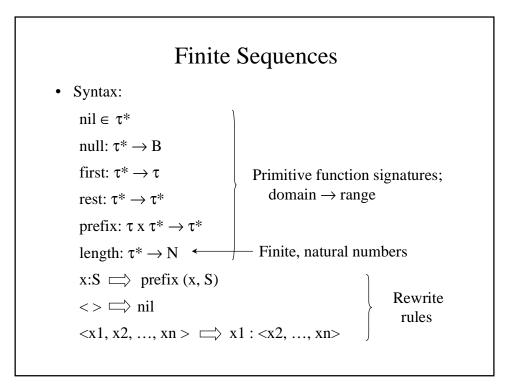


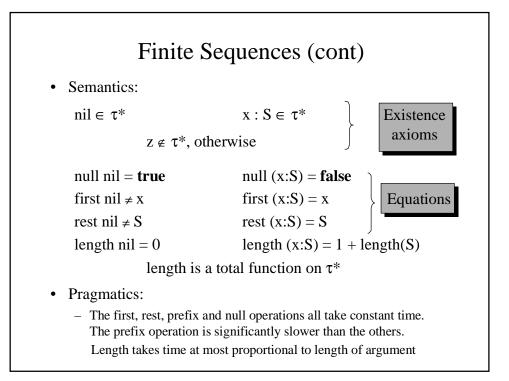












Operations on Sequences
Concatenation
cat: $\tau * x \tau * \rightarrow \tau *$
e.g. cat (<1,2> <3,4,5>) = <1,2,3,4,5>
• Reductions
sum: $R^* \rightarrow R$
e.g. sum(<1,2,3,4,5>) = 15
max: $R^* \rightarrow R$
e.g. max(<1,2,3,4,5>) = 5
• Mappings

Finite Sets

• Handout...

Higher Order Functions

- Defs:
 - *zero-order functions*: data in the traditional sense.
 - *first-order functions*: functions that operate on zero-order functions.

e.g. FIRST: $\tau^* \rightarrow \tau$ REST: $\tau^* \rightarrow \tau^*$

- second-order functions: operate on first order
 - e.g. map: $(D \to R) \to (D^* \to R^*) \quad \forall D, R \in \text{type}$ uncurried: $((D \to R) \times D^*) \to R^*$

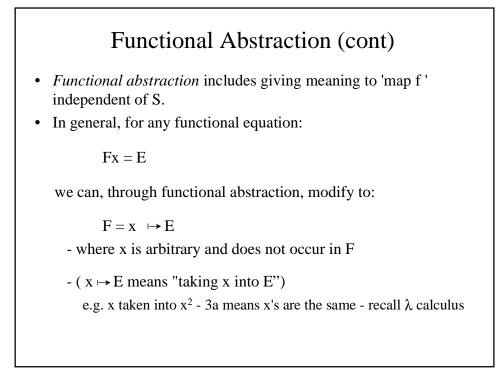
Higher Order Functions (cont)

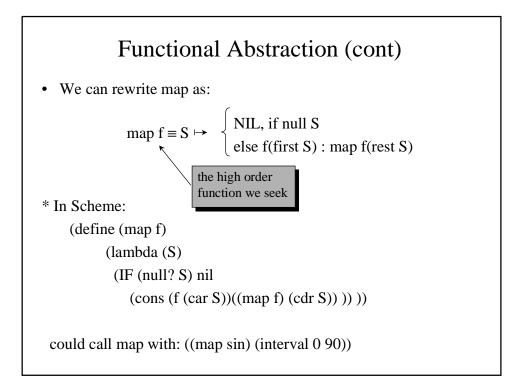
- In general, higher-order functions are those that can operate on functions of any order as long as types match.
 HOF's are usually polymorphic
- Higher-order functions can take other functions as arguments and produce functions as values.
- More defs:
 - *Applicative programming* has often been considered the application of first-order functions.
 - *Functional programming* has been considered to include higher-order functions: *functionals*.

Functional Programming

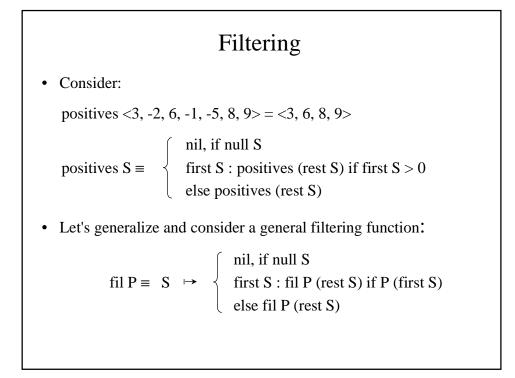
- Functional programming allows *functional abstraction* that is not supported in imperative languages, namely the definition and use of functions that take functions as arguments and return functions as values.
 - supports higher level reasoning
 - simplifies correctness proofs

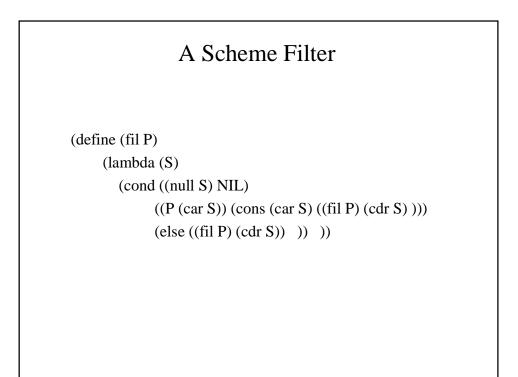
Functional Abstraction • For an arbitrary function, f, and sequence, S, we can define: $map f S = \begin{cases} NIL, if null S \\ else f(first S) : map f (rest S) \end{cases}$ - map is a two-argument function • map is applied to f • resulting function is applied to S • map f S signature: $map: [(D \to R) \times D^*] \to R^* \qquad - uncurried map: (D \to R) \to (D^* \to R^*) \forall D, R \in type \qquad - curried \end{cases}$ • map takes a function that maps from D to R and yields a function that maps from D to R.





	Map Archetype
Syntax: map: (1	$T \to U$) \to ($T^* \to U^*$), for all $T, U \in$ type
	s: nil = nil (x : S) = f x : map f S
	ics: equential implementations map f S takes linear time; on barallel implementations it takes constant time.
Prototyp	e: $map f \equiv S \mapsto \begin{cases} nil, if null S \\ else f(first S) : map f(rest S) \end{cases}$





Filter Archetype

Syntax:

fil: $(T \rightarrow B) \rightarrow (T^* \rightarrow T^*)$, for all $T \in$ **type**

Semantics:

fil P nil = nil fil P (x : S) = x : fil P S, if Px = **true**

fil P(x : S) = fil P S, if Px = false

Pragmatics:

with a sequential implementation fil P S takes linear time; with some parallel implementations it takes constant time.

Prototype:

fil
$$P \equiv S \mapsto \begin{cases} nil, if null S \\ first S : fil P (rest S) if P (first S) \\ else fil P (rest S) \end{cases}$$

Sequences and Sets

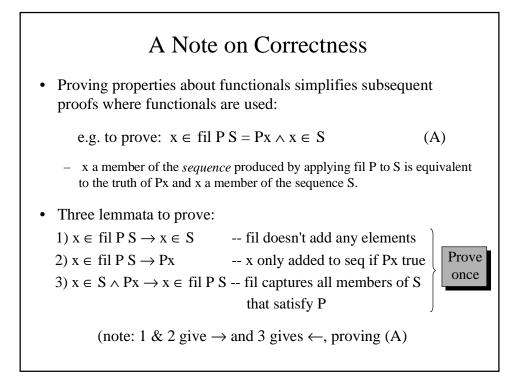
• A typical prototype for the finset archetype axiom:

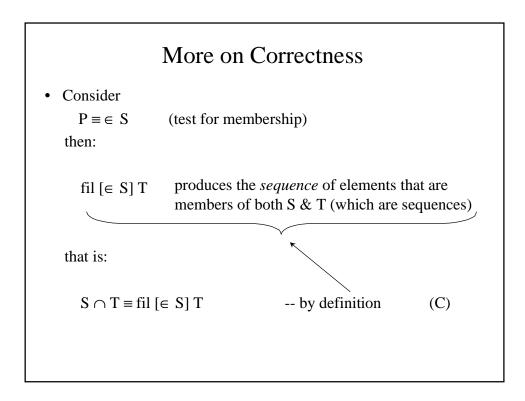
$$x \in (S \cap T) = x \in S \land \ x \in T$$

is:

$$S \cap T = \begin{cases} \phi \text{ if empty } S \\ \text{else adjoin (first } S, \text{ rest } S \ \cap T) \text{ if first } S \in T \\ \text{else rest } S \ \cap T \end{cases}$$

- \implies if we use sequences to represent sets then we must prove that we have true set operators.
- Proof can be lengthy, but HO functions can help.





Continue Correctness of Set Intersection

• So correctness of set intersection:

 $x \in (S \cap T) = x \in S \land x \in T$ -- from finite set archetype

can be demonstrated by:

$$\begin{aligned} \mathbf{x} \in (\mathbf{S} \cap \mathbf{T}) &= \mathbf{x} \in \operatorname{fil} \left[\in \mathbf{S} \right] \mathbf{T} & --\operatorname{from} \left(\mathbf{C} \right) \\ &= \left[\in \mathbf{S} \right] \mathbf{x} \land \mathbf{x} \in \mathbf{T} & --\operatorname{from} \left(\mathbf{A} \right) \\ &= \mathbf{x} \in \mathbf{S} \land \mathbf{x} \in \mathbf{T} & \Box \end{aligned}$$

• (This proof shows that a sequence implementation of a set can satisfy a set archetype)

Composition Archetype

Syntax:

°: $[(S \to T) \times (R \to S)] \to (R \to T)$, for all $R, S, T \in$ **type** that is, $(f \circ g): R \to T$ for $f: S \to T$ and $g: R \to S$

Semantics:

 $(f \circ g) x = f(g x)$

Pragmatics:

Composition takes the same time as the composed functions.

Prototype:

 $f \circ g \equiv x \mapsto f(g x)$

Construction Archetype

Syntax:

[;]: $[(S \to T) \times (S \to U)] \to [(S \to (T \times U))]$, for all *S*, *T*, *U* ∈ **type** that is, (f; g): $S \to (T \times U)$, for $f: S \to T$ and $g: S \to U$. $(f_1; f_2; ...; f_n) \quad (f_1; (f_2; ...; f_n))$

Semantics:

(f ; g) x = (f x, g x)

Pragmatics:

With sequential implementations, n-ary construction takes the sum of the times of the constructed functions. With some parallel implementations it takes the time of the slowest function.

Prototype:

 $f ; g \equiv x \mapsto (f x, g x)$

Haskell & ML: Interesting Features

- Type inferencing
- Freedom from side effects
- Pattern matching
- Polymorphism
- Support for higher order functions
- Lazy patterns / lazy evaluation
- Support for object-oriented programming

Type Inferencing

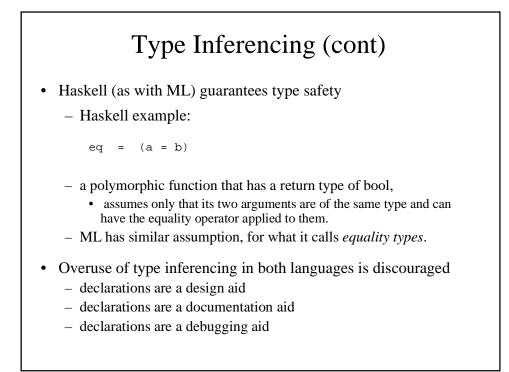
• Def: ability of the language to infer types without having programmer provide type signatures.

```
- SML e.g.:
```

```
fun min (a: real, b)
= if a > b
then b
else a
```

- type of a has to be given, but then that's sufficient to figure out

- type of b
- type of min
- What if type of a is not specified?
 - could be ints
 - could be bools...



Polymorphism

```
ML:

fun factorial (0) = 1
= | factorial (n) = n * factorial (n - 1)

ML infers factorial is an integer function: int -> int
Haskell:

factorial (0) = 1
factorial (n) = n * factorial (n - 1)

Haskell infers factorial is a (numerical) function: Num a => a -> a
```

Polymorphism (cont)

• ML:

fun mymax(x,y) = if x > y then x else y

```
- SML infers mymax is an integer function: int -> int
```

fun mymax(x: real ,y) = if x > y then x else y

- SML infers mymax is real

```
• Haskell:
```

mymax(x,y) = if x > y then x else y

- Haskell infers factorial is an Ord function