UNIVERSITETET
I OSLO

Logic Programming II & Revision

Arild B. Torjusen
aribraat@ifi.uio.no

Department of Informatics – University of Oslo

INF 3110/4110 - 2006

Outline

- Repetition
  - Facts, rules, queries and unification
  - Lists in Prolog
  - Different views of a Prolog program
- Today
  - Arithmetic in Prolog
  - Cut and negation
- Oblig 1
- Repetition ML

Facts, rules, queries and unification

- Remember: A declarative program admits two interpretations
  - Declarative interpretation, **What** is being computed.
  - Procedural interpretation, **How** the computation takes place
- A Prolog program consists of a sequence of *clauses*
- clauses are *facts* (H) or *rules* (H :- A_1,...,A_k)

  person(anne, sofia, martin, 1960) or child(X,Y) :- person(X,Z,Y,U))
- Declaratively, the rule H:- A_1 , A_2 is read as: ”H is implied by the conjunction A_1 , A_2”
- Procedurally, the rule H:- A_1 , A_2 is interpreted as ”To answer the query H, answer the conjunctive query A_1 , A_2”
- We initiate a computation by posing a *query* (|?- A_1,...,A_k)

  | ?- child(paul,Parent))
- For queries without variables we will get a yes/no answer.
- For queries with variables the result is the substitutions for (assignment of) the variables which will make the query true.
- The process of matching a query with facts and rules is called *unification*. The result of the unification is a *substitution*. (mgu = most general unifier)

Lists in Prolog

- [] : the empty list
- [a,b,c] : a list with three elements
- [a|[b,c]] : another way of writing [a,b,c]
- [a,b|[c]] : the same
- [X | Y] represents a list with first element X and tail Y
- the member predicate:

```
member(X, [X|Rest]).
member(X, [H | Tail]) :- member(X, Tail).
```

- the append predicate:

```
append([], Ys, Ys).
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
```

append

```
append([], Ys, Ys).                                   /* 1 */
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).   /* 2 */
```

```
|-? append([a,b],[c,d],Res)
append([X | Xs], Ys, [X | Zs]) :- append(Xs, Ys, Zs).
                                  {X=a, Xs=[b], Ys=[c,d], Res=[a|Zs]}
```

```
append([b], [c,d], Zs)
append([X1 | Xs1], Ys1, [X1 | Zs1]) :- append(Xs1, Ys1, Zs1).
                                  {X1=b, Xs1=[],Ys1=[c,d],Zs=[b|Zs1]}
```

```
append([], [c,d], Zs1)
append([], Ys2, Ys2)
                                  {Ys2=[c,d],Zs1=Ys2=[c,d]}
```

Res = [a|Zs]
= [a|[b|Zs1]]
= [a|[b|[c,d]]] = [a,b,c,d] .

Different views of a Prolog program

- **For testing:**

  | ?- member(wed, [mon, wed, fri]). yes

  | ?- append([a,b],[c,d],[a,b,c,d]) . yes

- **For computing:**

  | ?- member(X, [mon, wed, fri]).
  X = mon ? ; X = wed ?: X = fri ?; no

  | ?- append([a,b],[c,d],Zs) .
  Zs = [a,b,c,d] ? ;

  | ?- append(Xs, Ys, [a,b,c,d]).
  Xs = [],Ys = [a,b,c,d] ? ;
  Xs = [a], Ys = [b,c,d] ? ;
  Xs = [a,b],Ys = [c,d] ? ;
  ...

Outline

- Repetition
  - Facts, rules, queries and unification
  - Lists in Prolog
  - Different views of a Prolog program
- Today
  - Arithmetic in Prolog
  - Cut and negation
- Oblig 1
- Repetition ML

Arithmetic in Prolog

- Prolog programs presented so far were *declarative*: they admitted a dual reading as a formula
  - Operations of arithmetic are functional, not relational
- Arithmetic compromises Prolog's declarativeness
  - Solved in constraint logic programming languages

Arithmetic operators

- Built-in data structures:
  - Integers: 1,2,3,... (+, -, *, //)
  - Floating points: 2.3, 3.4456, 5.4e-13,... (+, -, *, /)
- Infix vs prefix notation*
  - 45+35
  - '+'(45,35)
- It is possible to have user-defined operators with specified priority, associativity, etc

*We will see later how to evaluate expressions

Arithmetic comparison relations

- Prolog allows comparison of ground arithmetic expressions (*gae*, i.e. expressions without variables). *gae*s have *values*
- Built-in comparison relations: <, =<, =:= (”equal”), =\= (”different”), >= and >
- Queries
  - | ?- 6*3 =:= 9*2.
    yes
  - | ?- 8 > 5+3.
    no
  - | ?- 34>=X+4.
    uncaught exception: error(instantiation_error,(>=)/2
- Note difference between
  - = (unifiability relation) 1+1=2 *gives* no, X = 1 *gives* X = 1
  - == (syntactic equality) 1+1 == 2 *gives* no , X == x *gives* no
  - \== (syntactic inequality) 1+1\==2 *gives* yes.
  - =:= (value equality) 1+1 =:= 2 *gives* yes
  - =\= (value inequality) 1+1 =\= 2 *gives* no

Example: ordered lists

```
ordered([]).
ordered([X]).
ordered([X,Y|Ys]) :- X =< Y, ordered([Y|Ys]).
```

- Queries
  - | ?- ordered([3,4,67,8]).
    no
  - | ?- ordered([3,4,67, 88]).
    yes
  - | ? - ordered([3,4,X,88]).
    {INSTANTIATION ERROR: 4=<_30 - arg 2}

Evaluation of arithmetic expressions

- We need to introduce a way to evaluate expressions
  - | ?- X=:=3+4. yields an error
  - | ?- X=3+4.
    X = 3+4
- Evaluation is done using ”is”
  - | ?- X is 3+4.
    X = 7
  - ”is” is a builtin predicate which has been defined as an operator for simpler syntax, we could also write:

    | ?- is(X,3+4).

    X = 7

Example: Factorial

```
factorial(0,1).
factorial(N,F) :- N>0, N1 is N-1, factorial(N1,F1),
                  F is N*F1.
```

- Queries
  - | ?- factorial(5,X).
    X = 120
    Yes

The following query gives, however, an error

- | ?- factorial(X,5).
  uncaught exception: error(instantiation_error,(>)/2)

"X>0" is not allowed!

Example: Length of lists

- An intuitive definition **but wrong**

```
length([],0).
length([_ | Ts], N+1) :- length(Ts,N).
```

- Query
  - | ?- length([3,5,56,7],X).

    X = 0+1+1+1+1

    Yes
- What's the problem?

Expressions are not automatically evaluated in Prolog!

Example: Length of lists

- A good definition

```
length([],0).
length([_ | Ts], N) :- length(Ts,M), N is M+1.
```

- Queries
  - | ?- length([3,5,56,7],X).
    X = 4
    Yes
  - | ?- length(X,5).
    X = [_,_,_,_,_]
    yes

Outline

- Repetition
  - Facts, rules, queries and unification
  - Lists in Prolog
  - Different views of a Prolog program
- Today
  - Arithmetic in Prolog
  - Cut and negation
- Oblig 1
- Repetition ML

cut

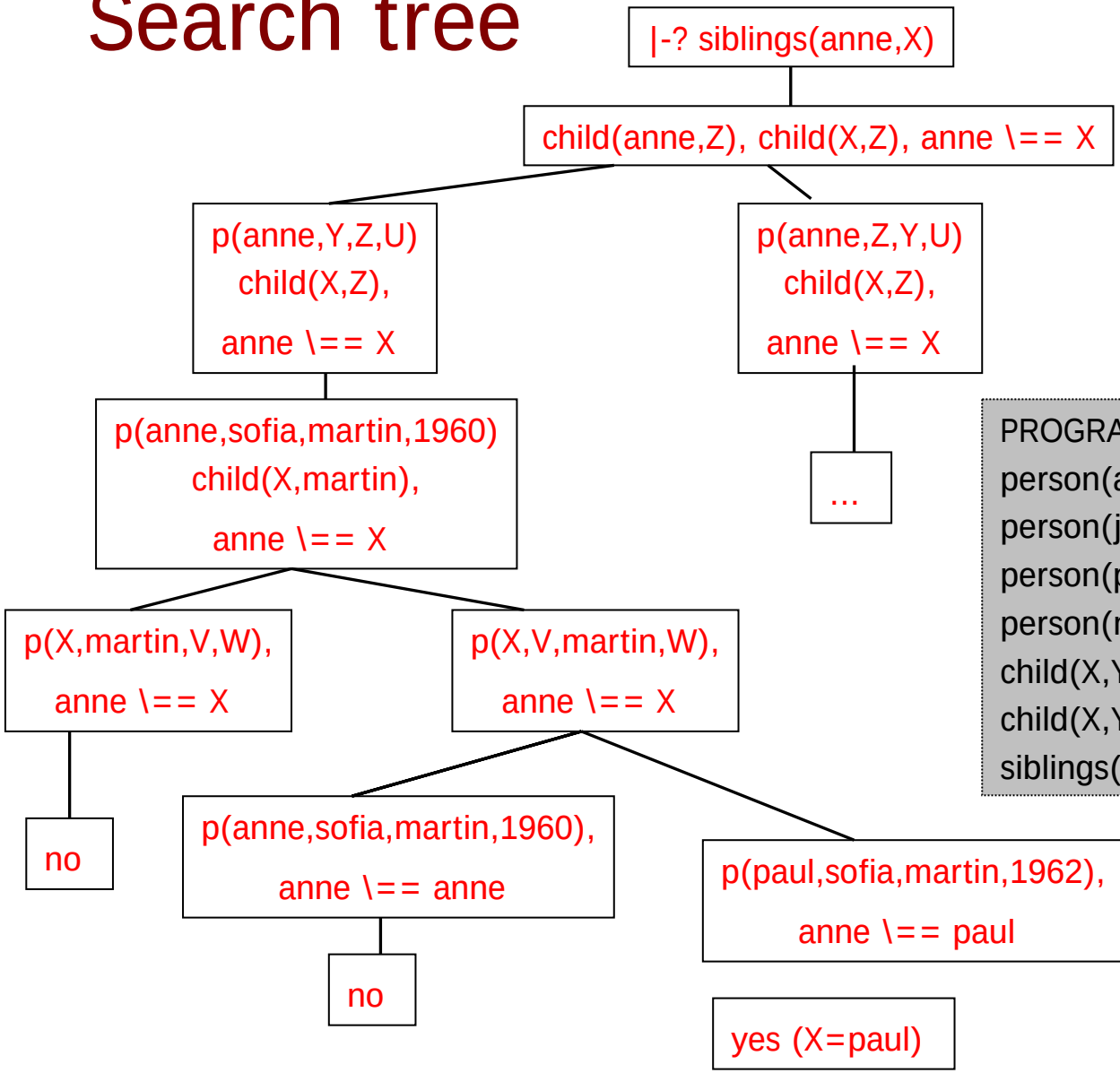
- *cut* is a built in system predicate which affects the procedural behaviour of a program
- its main function is to reduce the search space of Prolog computations by dynamically prunig the search tree.
- Ex:

```
p(s1) :- A1
...
p(si) :- B, !, C
...
p(sk) :- Ak
```

- When cut is encountered,
  1. all alternative ways of computing B is discarded.
  2. all computations of p(**t**) is discarded as backtrackable alternatives.

- cut gives more control to the programmer, but compromises the declarative reading of the Prolog programs and makes it difficult to see what will happen in the computatio.

Search tree



PROGRAM:

```
person(anne, sofia, martin, 1960).
person(john, sofia, george, 1965).
person(paul, sofia, martin, 1962).
person(maria, anne, mike, 1989).
child(X,Y) :- person(X,Z,Y,U).
child(X,Y) :- person(X,Y,Z,U).
siblings(X,Y) :- child(X,Z), child(Y,Z), X \== Y.
```

cut

- Recall the rsiblings rule.

```
rsiblings(X,Y) :- child(X,Parent1),
                  child(Y,Parent1),
                           X \== Y,
                  child(X,Parent2),
                  child(Y,Parent2),
                  Parent1 \== Parent2.
```

- With cut

```
rsiblings(X,Y) :- child(X,Parent1),
                  !,
                  child(Y,Parent1),
                           X \== Y,
                  child(X,Parent2),
                  child(Y,Parent2),
                  Parent1 \== Parent2.
```

Negation as failure

- Negation can be defined by cut.

```
not(X) :- X, ! , fail
not(_) .
```

- The built-in negation operator is \+

```
| ?- \+ person(arild,arild,lise,1969) .
yes
```

- The query \+ A succeeds if and only if the query A fails.
- Corresponds to our ”normal” notion of negation if the negated query always terminates and is ground. Consider negation of non-ground term X=1:

```
\+ (X=1)
no
```

The Zebra puzzle

Consider the following puzzle: There are five houses, each of a different color, and inhabitated by a man of a different nationality with a different pet, drink, and brand of cigarettes.

1. The Englishman lives in the red house.
2. The Spaniard owns the dog.
3. Coffee is drunk in the green house.
4. The Ukrainian drinks tea.
5. The green house is immediately to the right (your right) of the ivory house.
6. The Winston smoker owns snails.
7. Kools are smoked in the yellow house.
8. Milk is drunk in the middle house.
9. The Norwegian lives in the first house on the left.
10. The man who smokes Chesterfields lives in the house next to the man with the fox.
11. Kools are smoked in the house next to the house where the horse is kept.
12. The Lucky Strike smoker drinks orange juice.
13. The Japanese smokes Parliaments.
14. The Norwegian lives next to the blue house.

Now, who drinks water? Who owns the zebra?

```
solve:-  clues(Houses),   queries(Houses).

clues(Houses):-  house(A, Houses), colour(A, red), nationality(A, english),
house(B, Houses), nationality(B, spaniard), pet(B, dog),
house(C, Houses), colour(C, green), drink(C, coffee),
house(D, Houses), nationality(D, ukranian), drink(D, tea),
immed_to_right(Houses, E, F), colour(E, green), colour(F, ivory),
house(G, Houses), smoke(G, winston), pet(G, snails),
house(H, Houses), smoke(H, kools), colour(H, yellow),
middle(Houses, I), drink(I, milk),
first(Houses, J), nationality(J, norwegian),
next_to(Houses, K, L), smoke(K, chesterfields), pet(L, fox),
next_to(Houses, M, N), smoke(M, kools), pet(N, horse),
house(O, Houses), smoke(O, luckystrike), drink(O, orangejuice),
house(P, Houses), nationality(P, japanese), smoke(P, parliaments),
next_to(Houses, Q, R), nationality(Q, norwegian), colour(R, blue).

colour(house(C,_,_,_,_), C). nationality(house(_,N,_,_,_), N). pet(house(_,_,P,_,_), P). drink(house(_,_,_,D,_), D).
      smoke(house(_,_,_,_,S), S).

first(houses(X,_,_,_,_), X).  middle(houses(_,_,X,_,_), X).
immed_to_right(houses(L,R,_,_,_), R, L). immed_to_right(houses(_,L,R,_,_), R, L).
immed_to_right(houses(_,_,L,R,_), R, L). immed_to_right(houses(_,_,_,L,R), R, L).

next_to(Xs, X, Y):- immed_to_right(Xs, X, Y).
next_to(Xs, X, Y):- immed_to_right(Xs, Y, X).

house(X, houses(X,_,_,_,_)).house(X, houses(_,X,_,_,_)).house(X, houses(_,_,X,_,_)).house(X,
      houses(_,_,_,X,_)).house(X, houses(_,_,_,_,X)).

queries(Houses):-
  house(X, Houses), pet(X, zebra), nationality(X, Nationality1),
  write("The "), write(Nationality1), write(" owns the zebra"), nl,
  house(Y, Houses), drink(Y, water), nationality(Y, Nationality2),
  write("The "), write(Nationality2), write(" drinks water"), nl.
```

http://perso.orange.fr/colin.barker/sands.htm#Ex_14_1_iv

Outline

- Repetition
  - Facts, rules, queries and unification
  - Lists in Prolog
  - Different views of a Prolog program
- Today
  - Arithmetic in Prolog
- Oblig 1
- Repetition ML

Oblig 1 exercise 1

```
(* Helper function:  nth : int * 'a list -> 'a retrieves the nth element form a list. *)
(* Define an exception for cases that should not happen *) exception noSuchElement ;
fun nth (n,nil) = raise noSuchElement
fun nth (0,s::ss) = s
    | nth (n,s::ss) = nth((n-1),ss) ;

(* c = cursor  n = counter for how many digits we shall generate. xs = the rabitt sequenceso far.
If the cursor is at 1, we will always generate 2 digits, thus for some values we will get one digit
extra, we deal with that later in the toStr function We could have dealt with it in this function also. *)

      fun buildrabb(c,n,xs) = if (n>0) then
                              (case nth(c,xs) of 0 => buildrabb(c+1,n-1,xs@[1])
                                               | 1 => buildrabb(c+1,n-2,xs@[0,1])
                                               | _ => raise noSuchElement )
                              else xs ;
fun rabb(0) = []
   | rabb(1) = [0]
   | rabb(n) = buildrabb(0,n,[0])

fun toStr([],n) = ""
   | toStr(x::xs,n) = if n = 0 then "" else
                              case x of 0 => "0"^toStr(xs,n-1)
                                      | 1 => "1"^toStr(xs,n-1)
                                      | _ => raise noSuchElement ;

      fun rabSeq(n) = toStr(rabb(n),n) ;
```

```
(* Exercise 2 string compression *)


fun proc((c,t),[]) = []
   | proc((c,t),[x]) = if t=x then [(c+1,t)] else (c,t)::[(1,x)]
   | proc((c,t),x::xs) =
            case (c,t) of (0,_) => proc((1,x),xs)
                        | (n,z) => if z=x then
                                     proc((n+1,x),xs)
                                   else
                                     (c,t)::proc((0,#""),x::xs) ;

fun process(s) = proc((0,#" "),explode(s));

fun toString((c:int,t:char)) = ("^Int.toString(c)^","^Char.toString(t)^")";

fun processToString(s) = foldr (op^) "" (map toString (process(s))) ;
```

Paradigms: Overview

- Procedural/imperative Programming
  - A program execution is regarded as a sequence of operations manipulating a set of registers (programmable calculator)
- Functional Programming
  - A program is regarded as a mathematical function
- Object-Oriented Programming
  - A program execution is regarded as a physical model simulating a real or imaginary part of the world
- Constraint-Oriented/Declarative (Logic) Programming
  - A program is regarded as a set of equations

Many different languages

- Early languages
  - Fortran, Cobol, APL, ...
- Algol family
  - Algol 60, Algol 68, Pascal, ..., PL/1, ... Clu, Ada, Modula, Cedar/Mesa, ...
- Functional languages
  - Lisp, FP, SASL, ML, Miranda, Haskell, Scheme, Setl, ...
- Object-oriented languages
  - Simula, Smalltalk, Self, Cecil, ...
  - Modula-3, Eiffel, Sather, ...
  - C++, Objective C, ... Java

Languages are still evolving

- Object systems based on asynchronous calls
- Adoption of garbage collection
- Need of language support for
  - Concurrency primitives; abstract view of concurrent systems
  - Data-access; ex. XML
  - Security
  - Contracts; ex. in virtual organizations
- Domain-specific languages
- Aspect-oriented programming
- Concurrent and Distributed Systems (a lot to be done here!)
  - Network programming

Algol family and ML (Mitchell's chapter 5)

- Evolution of Algol family
  - Recursive functions and parameter passing
  - Evolution of types and data structuring
- ML: Combination of Lisp and Algol-like features
  - Expression-oriented
  - Higher-order functions
  - Garbage collection
  - Abstract data types
  - Module system
  - Exceptions
  - Type inference

Basic ML

- Interactive compiler: *read-eval-print*
- Basic Types: Booleans, Integers, Strings, Reals
- Compound Types:
  - *Unit ()*,
  - Tuples *(4, 5, "ha det!") : int * int * string;*
  - Records *{name="Anibal", hungry=true}: {name: string, hungry: bool};*
  - Lists: nil, x :: xs , [1,2,3] , 1 :: (2 :: (3 :: nil))
  - Operations on lists: append [1,2] @ [3,4]
- Value declarations
  - val <pat> = <exp> val myList = [1, 2, 3, 4]; val x::rest = myList;
  - Local declarations let val x = 2+3 in x*4 end;
- Function declarations
  - fun f(<pat>) = <expr> fun f (x,y) = x+y;
fun f x y = x + y
  - fn <pat> => <expr> fn x => x+1;
  - Multiple-clause definition: fun length (nil) = 0
| length (x::s) = 1 + length(s);

ML - cont

- Data-type declarations:
  - datatype color = red | yellow | blue;
  - datatype exp = Var of int | Const of int | Plus of exp*exp;
  - datatype tree = Leaf of int | Node of int*tree*tree;

  Recursive function on tree datatype:

```
fun sum (Leaf n) = n
  | sum (Node(n,t1,t2)) = n + sum(t1) + sum(t2);
```

  Function on exp type defined by a case expression

```
fun eval(e) = case e of  Var(n)    => n
                       | Const(n)  => n
                       | Plus(e1,e2) => eval(e1) + eval(e2) ;
```

- The keyword *type* can be used to define a type *abbreviation,* type int_pair = int * int ;
- ML is mostly functional, but has imperative constructs.

ML2

- Recursion
  - Standard vs tail-recursion
  - Two examples, reversing a list, and factorial. (also oblig1)
- Currying:
  - A function over pairs has type
    'a * 'b -> 'c
    while a curried function has type
    'a -> ('b -> 'c)
  - A curried function allows *partial application*: applied to its 1st argument (of type 'a), it results in a function of type 'b -> 'c
- Higher order functions (HOF, functionals)
  - fun map f nil = nil
    | map f (x::xs) = (f x) :: map f xs;
    val map = fn : ('a -> 'b) -> 'a list -> 'b list
  - map (fn x => x+1) [1,2,3];
    val it = [2,3,4] : int list
- Equality
  - Only certain types admits equality testing – (reals and functions do not!)

Something on ML Modules

- Signatures and structures are part of the standard *ML module system*
- An ML structure is a module, which is a collection of:
  - Types
  - Values
  - Structure declarations
- Signatures are module interfaces
  - Kind of ”type” for a structure

Types (Mitchell's chapter 6)

- Types are important in modern languages
  - Program organization and documentation
  - Prevent program errors
  - Provide important information to compiler
- Type safety
  - Relative type safety of languages
  - Compilee time vs. run-time type checking
- Type inference
  - Determine best type for an expression, based on known information about symbols in the expression
- Polymorphism
  - Single algorithm (function) can have many types
- Overloading
  - Symbol with multiple meanings, resolved at compile time

Another presentation

f(x) = 2+x equiv f = λx. (2+x) equiv f = λx. ((plus 2) x)

- Example
  - fun f(x) = 2+x;
  - (val f = fn x => 2+x ;)

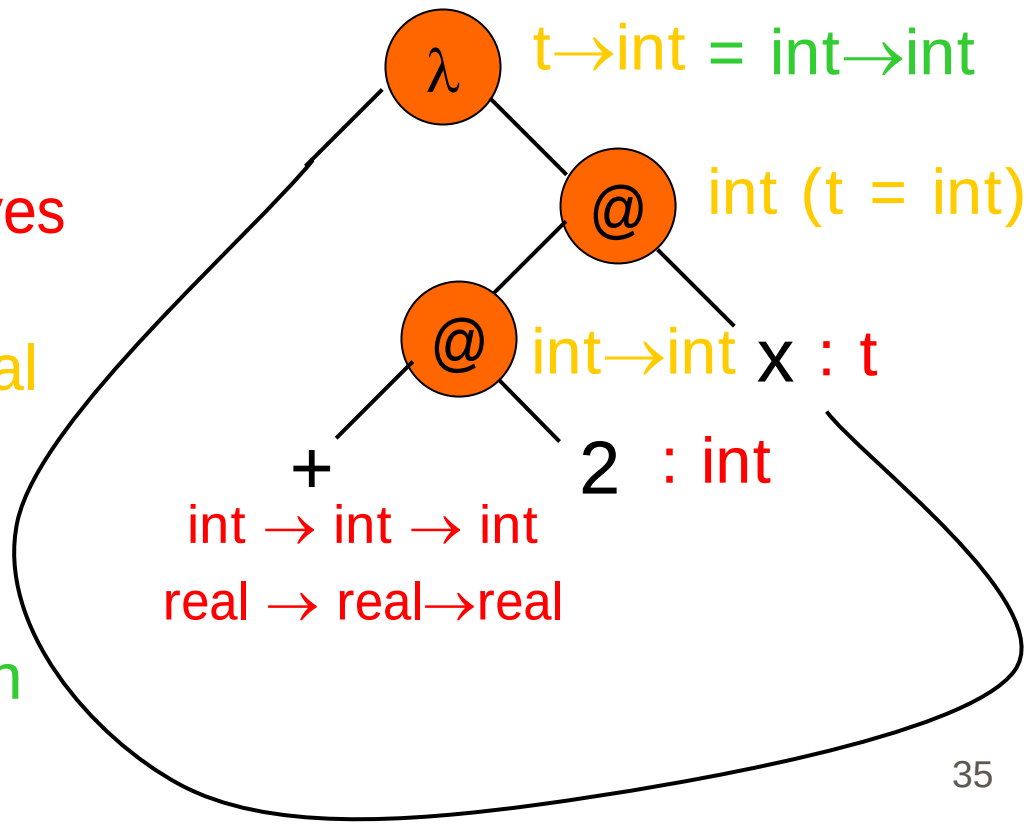
  val f = fn : int → int
- How does this work?

1. Assign types to leaves

2. Propagate to internal nodes and generate constraints

3. Solve by substitution

Graph for λx. ((plus 2) x)



Control (Mitchell's chapter 8)

- Structured Programming
  - "goto" considered harmful
- Exceptions in ML
  - A different kind of entity than types
  - "Structured" jumps that may return a value
  - Dynamic scoping of exception handler (The user knows best how to handle an exception)
  - Need to declare exceptions before use
  - Pattern matching is used to determine the appropriate handler (C++ uses type matching)
  - Can be used to handle errors or for better efficiency.