CS301
Session 25

Agenda

- Course evaluation
- Review

Why programming languages?

- Language influences thought
- P.L. features as tools for specifying computation
- Raise consciousness of language features
- Different programming styles: more powerful problem solving

Some language features

- Familiar:
  - Automatic storage management
  - Inheritance
- Strange:
  - Parametric polymorphism
  - First-class functions

Classifying languages

- Imperative, object-oriented, functional, logic programming and more
- Most are hybrids, e.g. Java is object-oriented and imperative
- Isolate features to understand what classifications mean

Formal semantics

- A taste of formal semantics will give you an idea of how we say precisely what a program will do

Impcore features

- Assignment: `(set x e)`
- Loop: `(while e1 e2)`
- Conditional: `(if e1 e2 e3)`
- Sequencing: `(begin e1 ... en)`
- Procedure: `(f e1 ... en)`

What is "imperative"?

- Computations work on a mutable store
- Order matters, e.g.

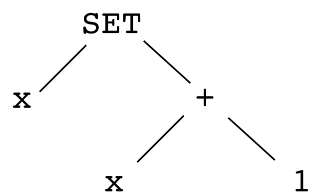
```
(begin (set x 1) (set x 2))
```

- is different from

```
(begin (set x 2) (set x 1))
```

Abstract syntax

- The **tree structure** of the language
- Data structure used by interpreters & compilers
- `(set x (+ x 1))`
  `x = x+1;`
  `x := x+1`



Free variables

- A variable name is an expression

  `x`

- but it means nothing in isolation; it is a *free variable*
- To give meaning to free variables, we use

Environments

Environments

- Environment: a mapping from names to meanings
- In Impcore meanings are *values*
- To bind a name to a value we write

  `(val x 2)`

- ...adding the mapping $x \mapsto 2$ to the current environment

Operational semantics

- Concise, precise guide to what the language *means*
- Specification for interpreter or compiler
- Supports proofs of language and program properties

How it works

- A set of inference rules specifies the behavior of a hypothetical *abstract machine*
- Use the rules to see how a particular expression is evaluated in a given context
- Reason about the system of rules to prove general properties of the object language

Applicative programming

- ...works by *applying* functions, not by mutating state
- The meaning of a name in a given scope doesn't change over time - no `set`
- Therefore we can have confidence in some simple laws.

The power of lambda

- At the top level, not interesting
- Used for local function definition, more interesting
- We can pass functions as parameters
- ...and - more interesting - return them as results

Closures

- We know what `v` means - a formal parameter that will be bound when the function is applied - but what does `t` mean in

```
(lambda (v) (lookup v t)))
```

?

Values for free variables

- Answer: it depends on the environment
- Evaluating a lambda-expression requires capturing the environment in a *closure*
- We don't write closures explicitly; the interpreter pairs the lambda-expression with the current environment:

$$\langle\langle\text{(lambda (v) (lookup v t))}, \{t \mapsto \text{'()}\}\rangle\rangle$$

Mutation and closures

- We can't bind variables to values in environments
- Because of assignment and closures, we bind variables to locations
- Bindings in a closure don't change, contents of locations can

Lambda

- Creates unnamed function

```
(lambda (x) (* x 3))
```

- ... the function that multiplies its argument by 3
- In

```
(lambda (x) (+ x y))
```

- ... `x` is *bound,* but `y` is *free*

Uses of lambda

- Define nested functions using letrec
- Pass functions as parameters
- Return functions as results
- Store them in data structures
- Polymorphic data structures
- Backtracking algorithms using continuations

Semantics of uScheme

- Top-level judgment:

$$\langle t, \rho, \sigma \rangle \rightarrow \langle \rho', \sigma' \rangle$$

- Expression evaluation judgment:

$$\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$$

What a rule means

- Operationally we read a rule as having inputs, possibly some subgoals, and outputs
  - Inputs: initial state of abstract machine
  - Subgoals: what the machine must do
  - Outputs: final state of abstract machine
- Note that metavariables x and x' and x_1 are all different!

Variables and assignment

- Variable lookup

$$\frac{x \in \text{dom}\,\rho \qquad \rho(x) \in \text{dom}\,\sigma}{\langle \text{VAR}(x), \rho, \sigma \rangle \Downarrow \langle \sigma(\rho(x)), \sigma \rangle}$$

- Assignment

$$\frac{x \in \text{dom}\,\rho \quad \rho(x) = l \quad \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{SET}(x, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma'\{l \mapsto v\} \rangle}$$

Let-binding

- Simultaneous binding

$$\frac{\begin{array}{c} l_1, \ldots, l_n \notin \text{dom}\,\sigma \\ \langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e, \rho\{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}, \sigma_n\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

Let* binding

- Sequential binding

$$\frac{\begin{array}{c} l_1, \ldots, l_n \notin \text{dom } \sigma \\ \langle e_1, \rho, \sigma \rangle \Downarrow \langle v_1, \sigma' \rangle \quad \rho_1 = \rho\{x_1 \mapsto l_1\} \quad \sigma_1 = \sigma'\{l_1 \mapsto v_1\} \\ \vdots \\ \langle e_n, \rho_{n-1}, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma'_{n-1} \rangle \quad \rho_n = \rho_{n-1}\{x_n \mapsto l_n\} \quad \sigma_n = \sigma'_{n-1}\{l_n \mapsto v_n\} \\ \langle e, \rho_n, \sigma_n \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{LETSTAR}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

Functions

- Lambdas evaluate to closures

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle \langle\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho \rangle\rangle, \sigma \rangle}$$

Functions

- Function applications

$$\frac{\begin{array}{c} l_1, \ldots, l_n \notin \text{dom } \sigma \\ \langle e, \rho, \sigma \rangle \Downarrow \langle \langle\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e_c), \rho_c \rangle\rangle, \sigma_0 \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e_c, \rho_c\{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n\}, \sigma_n\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{APPLY}(e, e_1, \ldots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

Type Systems

Static vs. dynamic checking

- Dynamic checking in uScheme:

```
-> (define appendfoo (l) (append 'foo l))
appendfoo
-> (appendfoo '(1 2))
error: car applied to non-pair foo in (car l1)
```

- Static checking in ML:

```
- fun appendfoo l = "foo" @ l;
! Toplevel input:
! fun appendfoo l = "foo" @ l;
!                   ^^^^^
! Type clash: expression of type
!   string
! cannot have type
!   'a list
```


Checking and interpreters

- Dynamic type checks: integrated with evaluation - a single-stage interpreter
- Static type checks: first phase of a two-stage interpreter

Static type checking: why?

- Not just to annoy novice programmers
- Support for serious programming
  - Catch mistakes at compile time and reduce dependency on completeness of testing
  - Document the intended behavior of programs
  - Define interfaces between modules
- Support for optimizing compilers

What Impcore types do

- In a well-typed Impcore program we know:
  - Every function (including primitives) receives the right number and type of actual parameters
  - Only Booleans are used for flow control (if and while)
- ...and we know this without ever running the program!

What they don't do

- We don't know if
  - there is division by 0
  - application of car or cdr to the empty list
  - illegal array indexing
  - infinite looping or recursion
  - wrong answers

Type system for Impcore

- Simple types: $\tau \;= \text{INT}|\text{BOOL}|\text{UNIT}|\text{ARRAY}(\tau)$
- Function types: $\tau_f = \tau_1 \times \ldots \times \tau_n \to \tau$
- Typing judgment for expressions: $\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau$
- Typing judgement for top-level items:

  $\langle t, \Gamma_\xi, \Gamma_\phi \rangle \to \langle \Gamma'_\xi, \Gamma'_\phi \rangle$
- Properties: deterministic, sound w.r.t. evaluation

Literals and variables

- Literals are numbers

$$\frac{}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{LITERAL}(v) : \text{INT}}$$

- Variable types are kept in the type environments

$$\frac{x \in \text{dom } \Gamma_\rho}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\rho(x)}$$

$$\frac{x \notin \text{dom } \Gamma_\rho \quad x \in \text{dom } \Gamma_\xi}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{VAR}(x) : \Gamma_\xi(x)}$$

Assignments

- To parameters

$$\frac{\begin{array}{l} x \in \text{dom } \Gamma_\rho \quad \Gamma_\rho(x) = \tau \\ \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau \end{array}}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{SET}(x, e) : \tau}$$

- To globals

$$\frac{\begin{array}{l} x \notin \text{dom } \Gamma_\rho \quad x \in \text{dom } \Gamma_\xi \\ \Gamma_\xi(x) = \tau \\ \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e : \tau \end{array}}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{SET}(x, e) : \tau}$$

Typing an if-expression

- The two arms must have the same type - why?

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_3 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{IF}(e_1, e_2, e_3) : \tau}$$

Typing a while-loop

- The value returned is the uninteresting "unit"

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \text{BOOL} \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_2 : \tau}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{WHILE}(e_1, e_2) : \text{UNIT}}$$

Typing a sequence

- Types 1 ... n-1 are uninteresting

$$\frac{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_1 : \tau_1 \ldots \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_n : \tau_n}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{BEGIN}(e_1, \ldots, e_n) : \tau_n}$$

Function application

- Using the function type environment

$$\frac{\Gamma_\phi(f) = \tau_1 \times \ldots \times \tau_n \rightarrow \tau \quad \Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash e_i : \tau_i}{\Gamma_\xi, \Gamma_\phi, \Gamma_\rho \vdash \text{APPLY}(f, e_1, \ldots, e_n) : \tau}$$

- We check that the actual parameters have the required types

Function definition

- Extending the function type environment

$$\frac{\Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \ldots \times \tau_n \rightarrow \tau\}, \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\langle \text{DEFINE}(f, (\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, e : \tau), \Gamma_\xi, \Gamma_\phi, \rightarrow \langle \Gamma_\xi, \Gamma_\phi\{f \mapsto \tau_1 \times \ldots \times \tau_n \rightarrow \tau\}}$$

- Notice how we *assume* the formals have the correct types while we are typing the body!

Top level value binding

- We extend the global type environment

$$\frac{\Gamma_\xi, \Gamma_\phi, \{\} \vdash e : \tau}{\langle \text{VAL}(x, e), \Gamma_\xi, \Gamma_\phi \rangle \rightarrow \langle \Gamma_\xi\{x \mapsto \tau\}, \Gamma_\phi \rangle}$$

Three common type constructors

- (First-class) functions
- Products
- Sums

First-class functions

- Type constructor $\longrightarrow$
  - Infix, two arguments: $\tau_1 \longrightarrow \tau_2$
- Formation rule:

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 \rightarrow \tau_2 \text{ is a type}}$$

Typing rules for functions

- Introduction

$$\frac{\Gamma\{x \mapsto \tau\} \vdash e : \tau'}{\Gamma \vdash \text{LAMBDA}(x : \tau, e) : \tau \to \tau'}$$

- Elimination

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \text{APPLY}(e_1, e_2) : \tau'}$$

Products (pairs)

- Constituent types need not be the same
- Variously, "tuple", "struct", "record"
- Can be used to model objects (in the OO sense)
- Formation

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 \times \tau_2 \text{ is a type}}$$

Typing rules for products

- Introduction

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

- Elimination

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2}{\Gamma \vdash \text{FST}(e) : \tau_1}$$

(and similarly for the second element)

An elegant elim rule

- Like a pattern match

$$\frac{\Gamma \vdash e : \tau_1 \times \tau_2 \quad \Gamma\{x_1 \mapsto \tau_1, x_2 \mapsto \tau_2\} \vdash e' : \tau}{\Gamma \vdash \text{LETPAIR}(x_1, x_2, e, e') : \tau}$$

Sum types

- A type that unions other types together
- Like C unions, but safer because you can always tell what's there
- Like simple ML datatypes (no recursion)
- Formation rule

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 + \tau_2 \text{ is a type}}$$

Typing rules for sums

- Introduction

$$\frac{\Gamma \vdash e : \tau_1 \quad \tau_2 \text{ is a type}}{\Gamma \vdash \text{LEFT}_{\tau_2}(e) : \tau_1 + \tau_2}$$

$$\frac{\Gamma \vdash e : \tau_2 \quad \tau_1 \text{ is a type}}{\Gamma \vdash \text{RIGHT}_{\tau_1}(e) : \tau_1 + \tau_2}$$

Typing rules for sums(2)

- Elimination: like case or switch

$$\frac{\begin{array}{l}\Gamma \vdash e : \tau_1 + \tau_2 \\ \Gamma\{x_1 \mapsto \tau_1\} \vdash e_1 : \tau \\ \Gamma\{x_2 \mapsto \tau_2\} \vdash e_2 : \tau\end{array}}{\Gamma \vdash \mathsf{case}\ e\ \mathsf{of}\ \text{LEFT}(x_1) \Rightarrow e_1 \mid \text{RIGHT}(x_2) \Rightarrow e_2 : \tau}$$

About type soundness

Why trust a type system?

- Given a complex enough type system, we might be unable to see whether it behaves reasonably
- Language designers prove *type soundness* both to increase trust and to be explicit about what guarantees the type system provides

What is type soundness?

- A kind of claim we make about the relationship between the typing rules and the evaluation rules
- Loosely, "well-typed programs don't go wrong"
- Sample corollaries:
  - Functions always receive the right number and kind of arguments
  - No array access is out of bounds (a more advanced kind of type system)

Machinery needed for soundness

- The meaning of a type $[\![\tau]\!]$ is a set of values
- Examples
  - $[\![\text{INT}]\!] = \{\text{NUMBER}(n) \mid n \text{ is an integer}\}$
  - $[\![\text{BOOL}]\!] = \{\text{BOOL}(\#\mathsf{t}), \text{BOOL}(\#\mathsf{f})\}$
- This gives us a notation for the set of things a well typed expression is allowed to evaluate to

Proper environments

- If Γ and ρ are typing and value environments, respectively, we say ρ agrees with Γ whenever, for every x in dom (Γ),

  1. x is also in dom (ρ), and
  2. $\rho(x) \in [\![\Gamma(x)]\!]$

A soundness claim

- If

  1. Γ and ρ are typing and value environments, and
  2. ρ agrees with Γ, and
  3. $\Gamma \vdash e : \tau$ and $\langle \rho, e \rangle \Downarrow v$,

  then $v \in [\![\tau]\!]$

Limitations of monomorphic typing

- Example from typed Impcore: list processing functions

Polymorphism

- Introduce polymorphic type system with static type checking
- Now we can write one version of `length` with type

  $\forall \alpha . \alpha$ `list` $\rightarrow$ `int`

  ```
  (forall ('a) (function ((list 'a)) int))
  ```

- This will be flexible enough to type a lot of the programs we want - almost a "sweet spot"
- ...but terribly verbose and impossible to use

Why?

- Why torture ourselves with this type system?
- To motivate *type inference* as in ML and related languages
- The real "sweet spot": polymorphic type system, plus type inference, yields a terse, flexible language with robust guarantees suitable for production programming
- Used in ML, OCaml, Haskell, etc. etc.

Type variables

- A new kind of variable that stands for an unknown type
- Actual types are supplied by *type instantiation,* a.k.a. *type application*
- Type variables are bound in types by ∀(abstractly),

  or `forall` (concretely)
- Bound in expressions by TYLAMBDA (abstractly), or `type-lambda` (concretely)

Idea: lambda for types

- You've seen this before: Java/C++ generics
- Quantified types: $\forall\alpha_1, \ldots, \alpha_n \,.\, \tau$

  `(forall ('a1 ... 'an) type)`
- Type abstraction: $\text{TYLAMBDA}(\alpha_1, \ldots, \alpha_n, e)$

  `(type-lambda ('a1 ... 'an) exp)`
- Type application: $\text{TYAPPLY}(e, \tau_1, \ldots, \tau_n)$

  `(@ exp type1 ... typen)`

Quantified types

```
 -> length
<procedure> : (forall ('a) (function ((list 'a)) int))

-> cons
<procedure> : (forall ('a) (function ('a (list 'a)) (list 'a)))

-> car
<procedure> : (forall ('a) (function ((list 'a)) 'a))

-> cdr
<procedure> : (forall ('a) (function ((list 'a)) (list 'a)))

-> '()
() : (forall ('a) (list 'a))
```

Type instantiation

```
 -> (val length-int (@ length int))
length-int : (function ((list int)) int)

-> (val length-bool (@ length bool))
length-bool : (function ((list bool)) int)

-> (val nil-bool (@ '() bool))
() : (list bool)
```

Instantiation *substitutes* actual types for type variables

Type abstraction

```
-> (val-rec (forall ('a) (function ((list 'a)) int))
     len (type-lambda ('a)
       (lambda (((list 'a) l))
         (if ((@ null? 'a) l) 0
            (+ 1 ((@ len 'a) ((@ cdr 'a) l)))))))
len : (forall ('a) (function ((list 'a)) int))
-> (@ len int)
<procedure> : (function ((list int)) int)
-> ((@ len int) '(1 2 3))
3 : int
```


Lambda for types

- Remember the basic idea: abstract over types
- Quantified types: $\forall \alpha_1, \ldots, \alpha_n \,.\, \tau$

  `(forall ('a1 ... 'an) type)`
- Type abstraction: $\text{TYLAMBDA}(\alpha_1, \ldots, \alpha_n, e)$

  `(type-lambda ('a1 ... 'an) exp)`
- Type application: $\text{TYAPPLY}(e, \tau_1, \ldots, \tau_n)$

  `(@ exp type1 ... typen)`

Type expressions versus types

- Our language of types is getting fairly complex:

```
datatype tyex = TYCON  of name (* constructor *)
              | TYVAR  of name (* type variable *)
              | CONAPP of tyex * tyex list
                     (* apply a constructor *)
              | FORALL of name list * tyex
                     (* polymorphic type *)
```

- Type constructors are things like list, function, pair, and so on
- Constructors are *applied* to other types to obtain types, e.g. `(list int)`
- Polymorphic types are not applied; but the values they describe are applied to types

Classifying type expressions

- Instead of having a set of "type-formation" rules like

$$\frac{\tau_1 \text{ and } \tau_2 \text{ are types}}{\tau_1 \to \tau_2 \text{ is a type}}$$

  we have a kind system "on top of" our type system, to classify our type expressions.
- This is used to ensure that types are well formed, e.g. to rule out something like:

  `(define (list list) foo () 0)`

Kinds

- A *kind environment* classifies our types:

  $\texttt{int} :: *, \texttt{bool} :: *, \texttt{unit} :: *$

  and constructors:

  $\texttt{list} :: * \Rightarrow *, \rightarrow :: * \times * \Rightarrow *, \texttt{array} :: * \Rightarrow, \ldots$

- To extend the language we can add to the kind environment:

  $\texttt{pair} :: * \times * \Rightarrow *, \texttt{sum} :: * \times * \Rightarrow *$

Using kinds

- Kinding rules tell when type expressions are well formed

$$\frac{\mu \in \text{dom } \Delta}{\Delta \vdash \text{TYCON}(\mu) :: \Delta(\mu)}$$

- E.g.,

$$\frac{\texttt{list} \in \text{dom } \Delta}{\Delta \vdash \text{TYCON}(\texttt{list}) :: * \Rightarrow *}$$

Constructor applications

- This kinding rule is the twin of the typing rule for function application:

$$\frac{\begin{array}{c}\Delta \vdash \tau :: \kappa_1 \times \ldots \times \kappa_n \Rightarrow \kappa \\ \Delta \vdash \tau_1 :: \kappa_1 \ldots \Delta \vdash \tau_n :: \kappa_n\end{array}}{\Delta \vdash \text{CONAPP}(\tau, [\tau_1, \ldots, \tau_n]) :: \kappa}$$

- We can use this rule to check that `(list int)` is a properly formed type.

A special case: tuples

- The tuple type constructor has variable arity:

$$\frac{\Delta \vdash \tau_i :: *, \quad 1 \leq i \leq n}{\Delta \vdash \text{CONAPP}(\text{TYCON}(\texttt{tuple}, [\tau_1, \ldots, \tau_n]) :: *}$$

Quantified types

- Where the polymorphism action is:

$$\frac{\Delta\{\alpha_1 :: *, \ldots, \alpha_n :: *\} \vdash \tau :: *}{\Delta \vdash \text{FORALL}(\langle \alpha_1, \ldots, \alpha_n \rangle, \tau) :: *}$$

- This rule is the "twin" of the typing rule for functions!
- We look up type variables in the kind environment

$$\frac{\alpha \in \text{dom } \Delta}{\Delta \vdash \text{TYVAR}(\alpha) :: \Delta(\alpha)}$$

An important restriction

- Type variables must have kind *
  ...so we can't quantify over, say, type constructors
- We can say "for any type", but not "for any type constructor"
- Other type systems (e.g. Haskell's) relax this restriction

The uScheme type system

- The typing rules are much like typed Impcore, but
  - only one type environment
  - a kind environment is needed for type constructors and type variables
  - no special rules for constructors like array

Typing let-binders

- Let and let*, no letrec

$$\frac{\begin{array}{l}\Delta, \Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n \\ \Delta, \Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau\end{array}}{\Delta, \Gamma \vdash \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau}$$

- We view let* as syntactic sugar for nested let

$$\frac{\Delta, \Gamma \vdash \text{LET}(\langle x_1, e_1 \rangle, \text{LETSTAR}(\langle x_2, e_2, \ldots, x_n, e_n \rangle, e)) : \tau, \quad n > 0}{\Delta, \Gamma \vdash \text{LETSTAR}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau}$$

$$\frac{\Delta, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \text{LETSTAR}(\langle \rangle, e) : \tau}$$

Typing lambda

- We check that the declared parameter types are well formed

$$\frac{\Delta \vdash \tau_i :: *, 1 \leq i \leq n \quad \Delta, \Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Delta, \Gamma \vdash \textsc{lambda}(\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, e) : \tau_1 \times \ldots \times \tau_n \to \tau}$$

- Then assume that the variables have these types while type-checking the body.

Typing APPLY

- Same as for Impcore, except that the type of the function is no longer stored in a function environment

$$\frac{\Delta, \Gamma \vdash e_i : \tau_i, 1 \leq i \leq n \quad \Delta, \Gamma \vdash e : \tau_1 \times \ldots \times \tau_n \to \tau}{\Delta, \Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \tau}$$

Typing TYLAMBDA

- Instead of putting new ordinary variables in the type environment, we put new type variables in the kind environment:

$$\frac{\Delta\{\alpha_1 :: *, \ldots, \alpha_n :: *\}, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \textsc{tylambda}(\alpha_1, \ldots, \alpha_n, e) : \forall \alpha_1 \ldots \alpha_n . \tau}$$

Typing TYAPPLY

- We check that the applied term has a polymorphic type and that the arguments are all types

$$\frac{\Delta \vdash \tau_i :: *, 1 \leq i \leq n \quad \Delta, \Gamma \vdash e : \forall \alpha_1 \ldots \alpha_n . \tau}{\Delta, \Gamma \vdash \textsc{tyapply}(e, \tau_1, \ldots, \tau_n) : \tau[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]}$$

- The resulting type is constructed by *substituting* the arguments for the type variables in the body of the polymorphic type.

Typing VAL and VAL-REC

- Note the different handling of the environment!

$$\frac{\Delta, \Gamma \vdash e : \tau}{\Delta, \Gamma \vdash \text{VAL}(x, e) \rightarrow \Gamma\{x \mapsto \tau\}}$$

$$\frac{\Delta, \Gamma\{x \mapsto \tau\} \vdash e : \tau}{\Delta, \Gamma \vdash \text{VAL-REC}(x, \tau, e) \rightarrow \Gamma\{x \mapsto \tau\}}$$

Evaluation

- No extra work is needed to interpret typed uScheme! After type checking, types are "thrown away" and the evaluator works as before - except for error handling.
- But we need to specify the semantics of the new constructs - type application and abstraction, and VAL-REC
- And we need to be careful with VAL!

Type application & abstraction

- Forget the types

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{TYAPPLY}(e, \tau_1, \ldots, \tau_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

$$\frac{\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}{\langle \text{TYLAMBDA}(\langle \alpha_1, \ldots, \alpha_n \rangle, e), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle}$$

VAL - a pitfall

- Suppose VAL doesn't always create a new binding

```
> uscheme
-> (val x 1)
1
-> (define f (n) (+ x n))
f
-> (f 2)
3
-> (val x '(a b))
(a b)
-> (f 2)
error: in (+ x n), expected an integer, but got (a b)
```

VAL pitfall (2)

- Since typed uScheme has no run-time type checking, VAL must create a *new* variable, not assign to an old one!

```
> tuscheme
-> (val x 1)
1 : int
-> (define int f ((int n)) (+ x n))
f : (function (int) int)
-> (f 2)
3 : int
-> (val x '(a b))
(a b) : (list sym)
-> (f 2)
3 : int
```


From typed uScheme to uML

Pure functional programming

- a.k.a. *applicative* programming
- Negatively, lack of mutation & related features (crudely: "no side effects")
- Positively, *referential transparency:* the value of an expression depends only on the values of its subexpressions.
  - In particular, the value doesn't depend on the context of the expression!

Benefits of r.t.

- Simple semantics
- Predictability and provability of programs
- Easy compiler optimizations
- Easy thread safety
- ...

μML

- ML proper does have assignment, but μML does not.
- μML has output and error exit (imperative), and loops and sequencing (only interesting in the presence of imperative features).
- So the only side effects are output and early termination.

Abstract syntax of μML

- Same as μScheme,
  - *but leaving out* SET (assignment), WHILE (loops)
  - *and adding in* VALREC as in typed μScheme
- Values are the same, but subject to a type system
  - numbers, booleans, and symbols
  - pairs
  - closures and primitive functions

Operational semantics

- No locations - why not?
- The only result of expression evaluation is a value
- The only result of top-level evaluation is a new environment
- Rule for "begin" shows we don't care about order

Contrasting begin rules

- μML

$$\frac{\begin{array}{l}\langle e_1, \rho\rangle \Downarrow v_1 \\ \langle e_2, \rho\rangle \Downarrow v_2 \\ \vdots \\ \langle e_n, \rho\rangle \Downarrow v_n\end{array}}{\langle \mathrm{BEGIN}(e_1, e_2, \ldots, e_n), \rho\rangle \Downarrow v_n} \qquad \frac{}{\langle \mathrm{BEGIN}(), \rho\rangle \Downarrow \mathrm{NIL}}$$

- μScheme

$$\frac{\begin{array}{l}\langle e_1, \rho, \sigma_0\rangle \Downarrow \langle v_1, \sigma_1\rangle \\ \langle e_2, \rho, \sigma_1\rangle \Downarrow \langle v_2, \sigma_2\rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1}\rangle \Downarrow \langle v_n, \sigma_n\rangle\end{array}}{\langle \mathrm{BEGIN}(e_1, e_2, \ldots, e_n), \rho, \sigma_0\rangle \Downarrow \langle v_n, \sigma_n\rangle} \qquad \frac{}{\langle \mathrm{BEGIN}(), \rho, \sigma\rangle \Downarrow \langle \mathrm{BOOL}(\texttt{\#f}), \sigma\rangle}$$

Closures

- As in Scheme, a lambda expression evaluates to a closure containing the current environment.
- To apply a lambda we use the environment when evaluating the body:

$$\frac{\begin{array}{l}\langle e, \rho\rangle \Downarrow \langle\!\langle \text{LAMBDA}(\langle x_1, \ldots, x_n\rangle, e_c), \rho_c\rangle\!\rangle \\ \langle e_1, \rho\rangle \Downarrow v_1 \ldots \langle e_n, \rho\rangle \Downarrow v_n \\ \langle e_c, \rho_c\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\}\rangle \Downarrow v\end{array}}{\langle \text{APPLY}(e, e_1, \ldots, e_n), \rho\rangle \Downarrow v}$$

Recursion (1)

- Up to now we handled semantics of recursion by early binding and mutation to install a circular reference in an environment
- No mutation - so we simply state the requirement for a circular reference
- We guarantee that we can do it by restricting recursion to lambda!

Recursion (2)

- Simple, but tricky: we create an environment that contains references to itself!

$$\frac{\begin{array}{l}e_1, \ldots, e_n \text{ are all LAMBDA epressions} \\ \rho' = \rho\{x_1 \mapsto v_1, \ldots, x_n \mapsto v_n\} \\ \langle e_1, \rho'\rangle \Downarrow v_1 \quad \ldots \quad \langle e_n, \rho'\rangle \Downarrow v_n \\ \langle e, \rho'\rangle \Downarrow v\end{array}}{\langle \text{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e), \rho\rangle \Downarrow v}$$

Recursion (3)

- Implementation uses a simple trick: an ML function captures the environment in an ML closure

```
datatype  value = NIL
          ...
          | CLOSURE   of lambda * (unit -> value env)
fun eval(e, rho) = let fun ...
    | ev(LETX (LETREC, bs, body)) =
        let fun makeRho' () =
          let fun step ((n, e), rho) =
           (case e
              of LAMBDA l => bind(n, CLOSURE (l, makeRho'), rho)
                      | _ => raise RuntimeError "non-lambda in letrec")
          in  foldl step rho bs
          end
         in  eval(body, makeRho'())
         end
```

- Shallow embedding again!

Recursion for lambda only!

- In µScheme:

```
(letrec ((odd-even (list2
  (lambda (n) (let ((even (cadr odd-even)))
    (if (< n 0) (even (+ n 1))
        (if (> n 0) (even (- n 1)) #f))))
  (lambda (n) (let ((odd (car odd-even)))
    (if (< n 0) (odd (+ n 1))
        (if (> n 0) (odd (- n 1)) #t)))))))

(list2 ((car odd-even) 3) ((cadr odd-even) 4)))
(#t #t)
```

- In µML:

```
run-time error: non-lambda in letrec
```


Type system

- Once again we have type expressions of
  - variables α
  - constructors μ
  - applications of constructors $(\tau_1,\ldots,\tau_n)\tau$
    - note *postfix* notation
  - quantification $\forall\ \tau_1,\ldots,\tau_n.\tau$ but quantification is restricted to the top level or outside
- No kinds - the programmer never writes a type

Type schemes

- In typed µScheme quantifiers are fully general:

```
-> (val not-too-poly
     (lambda (((forall ('a) (list 'a)) nil))
      ((@ pair (list int) (list bool))
        ((@ cons int) 1 (@ nil int))
        ((@ cons bool) #t (@ nil bool)))))
not-too-poly : (function ((forall ('a) (list 'a)))
                         (pair (list int) (list bool)))
-> (not-too-poly '())
((1) #t) : (pair (list int) (list bool))
```

- Not allowed in µML:

```
-> (val too-poly (lambda (nil)
                   (pair (cons 1 nil) (cons #t nil))))
  type error: Cannot unify int and bool
```


Type system

- We can give a straightforward - but nondeterministic! - set of typing rules.
- Rules for if, begin, apply, etc. are familiar from typed µScheme (but no kind environment needed)
- Rules for variables and lambda are nondeterministic
- Rules for let/letrec infer type schemes

Type inference

- The issue is how to turn *nondeterministic* rules into a *deterministic* type inference algorithm
- The algorithm is presented in terms of inference rules that "return" a substitution as well as a type!
- Unification is the way we find substitutions

Type inference judgment

- In $\theta\Gamma \vdash e : \tau$, the substitution theta and the type tau are outputs
- The type may contain type variables
- The typing context contains *type schemes*

Type inference for APPLY

- Type checking:

$$\frac{\Gamma \vdash e : \tau_1 \times \ldots \times \tau_n \to \tau \qquad \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n}{\Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \tau}$$

- Inference:

$$\frac{\theta\Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \qquad \theta'(\hat{\tau}) = \theta'(\tau_1 \times \ldots \times \tau_n \to \alpha), \text{where } \alpha \text{ is fresh}}{(\theta' \circ \theta)\Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \theta'\alpha}$$

Operational interpretation

- Infer types $\hat{\tau}, \tau_1, \ldots, \tau_n$ for $e, e_1, \ldots, e_n$, yielding substitution θ
- Pick fresh type var α and unify $\hat{\tau}$ with $\tau_1, \ldots, \tau_n \to \alpha$, yielding θ'
- Answer type is $\theta'\alpha$, answer substitution is $\theta' \circ \theta$

Soundness

- *Soundness* of the type inference rules means that if we infer a type for e using the type inference system, then e has that type according to the type checking system.
- *Soundness* can be proved by induction on the structure of a type inference.

Example

- Let's infer a type for `(car '(1))` APPLY(PRIM(car),LITERAL(PAIR(NUM(1),NIL)))
- Type scheme $\forall\alpha.\alpha\ \texttt{list} \to \alpha$ for car is found in environment, and we take its most general instance, or $\alpha\ \texttt{list} \to \alpha$; for the literal we use the rule on p. 236 to get $\texttt{int list}$; our substitution is still "empty", or id.
- So now we have types for the function and for its argument, and we want to match them up.

Example (cont'd)

- We pick a fresh type variable β and unify $\alpha\ \texttt{list} \to \alpha$ with $\texttt{int list} \to \beta$; the answer substitution is $\theta' = \{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{int}\}$
- So the answer type is $\theta'\beta = \texttt{int}$
- and the answer substitution is $\theta' \circ \mathrm{id} = \{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{int}\}$
- Notice how unification implicitly filled in the type application `(@ car int)`

Type inference for variables

- The typing rule for variables is nondeterministic:

$$\frac{\Gamma(x) = \sigma \qquad \tau <: \sigma}{\Gamma \vdash x : \tau}$$

- To make it algorithmic, we use the *most general instance* of the type scheme:

$$\frac{\Gamma(x) = \sigma \quad \tau = \mathsf{freshinstance}(\sigma)}{\Gamma \vdash x : \tau}$$

Most general instance

- If σ is a type scheme and τ is a most general instance of σ, what could τ be?
- Example: σ is ∀α,β.α×β→(α×β) list

  could τ be $\beta_1\times\beta_2\to(\beta_1\times\beta_2)$ list?
  how about $\beta_1\times\beta_1\to(\beta_1\times\beta_1)$ list?
  int×bool→(int×bool) list?

Type inference for lambda

- Again the typing rule is nondeterministic:

$$\frac{\Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \text{LAMBDA}(\langle x_1, \ldots, x_n\rangle, e) : \tau_1 \times \ldots \times \tau_n \to \tau}$$

- We introduce fresh type variables:

$$\frac{\begin{array}{l}\alpha_1, \ldots, \alpha_n \text{ are fresh} \\ \Gamma' = \Gamma\{x_1 \mapsto \forall.\alpha_1, \ldots, x_n \mapsto \forall.\alpha_n\} \\ \theta\Gamma' \vdash e : \tau\end{array}}{\theta\Gamma \vdash \text{LAMBDA}(\langle x_1, \ldots, x_n\rangle, e) : \theta\alpha_1 \times \ldots \times \theta\alpha_n \to \tau}$$

Operational interpretation

- Pick n fresh type variables and form type schemes $\forall.\alpha_i$
- Bind the x_i to $\forall.\alpha_i$ to form the new typing environment Γ'
- Infer a type τ for e in Γ', yielding substitution θ
- The answer substitution is θ and the answer type is $\theta\alpha_1\times\ldots\times\theta\alpha_n\to\tau$

Example

- Let's infer a type for `(lambda (x) (+ x 1))`
  LAMBDA(<x>,APPLY(PRIM(+),VAR(x),LIT(NUM(1)))
- Pick a fresh type variable α and bind x to ∀.α
- Infer a type for the body in the new environment
  - Use the rule for APPLY

Example (cont'd)

- Environment: $\{x \to \forall.\alpha\}$
- Infer types for PRIM(+), VAR(x), and LIT(NUM(1)), getting $int \times int \to int$, α, and int; the substitution is $\theta = id$
- Pick a fresh type variable β, and unify $int \times int \to int$ with $\alpha \times int \to \beta$, yielding substitution $\theta' = \{\alpha \mapsto int, \beta \mapsto int\}$
- Answer: $\theta'\beta = int$, and $\theta' \cdot \theta = \theta'$

Example (cont'd)

- Now we have typed the body of the lambda, so the answer substitution is θ', which is $\{\alpha \mapsto int, \beta \mapsto int\}$, and the answer type is $\theta'\alpha \to int$, which is $int \to int$.
- In this example the algorithm has "filled in" the unstated type of the formal parameter x in `(lambda (x) (+ x 1))`

Free variables

- The free type variables of a type scheme are those not bound by $\forall$
- For instance, in $\forall\alpha.\alpha \to \beta$, β is free (and α is bound)
- How about in $\forall.\alpha$?

Generalization

- To type let-binding, we generalize an inferred type *t* to create a type scheme, by "closing over" the variables that are free in *t*, but not over the variables free in the typing environment.
- E.g., generalize($\alpha \to \beta$,$\{x \mapsto \forall.\alpha\}$) is $\forall\beta.\alpha \to \beta$

Smalltalk

- Smalltalk: the original OO language
- *All values* in Smalltalk are objects, even numbers and booleans
- Other than message send (or method invocation) control flow mediated by boolean and block objects
- Blocks are closures and can be recursively defined at the top level

Object-oriented programming

- Language constructs: *objects* and *classes*
- Mechanisms: *inheritance* and *dynamic dispatch*
- Principles: *data encapsulation* and *code re-use*

Related languages

- Precursor: Simula
- Languages with OO features: CLOS, C++, OCaml, Eiffel, Python, Java, C#, even Visual Basic, many others
- OO is the language paradigm *du jour*

What is an object?

- An entity that responds to *messages* by changing its state and/or *answering* with a value
- An object is represented by a collection of
  - instance variables (private) that constitute its *state*
  - methods (public) that specify its response to messages
- Arguably, objects alone are enough for "pure" object-oriented programming

Adding classes

- Objects provide encapsulation and message handling
- Classes add *code re-use:* all members of the same class share the same methods
- Again, arguably we could stop there and have a meaningful OO language

Adding inheritance

- Inheritance creates a potentially complex web of code reuse
- Mechanisms: *subclassing* and *dynamic dispatch*
- Subclassing is *transitive*
- A subclass inherits the instance variables and methods of its superclass(es)
- A subclass may override (redefine) an inherited method

Dynamic dispatch

- How a message is handled is determined at runtime:
  - If there is a method defined in the receiver's class for the message, use it
  - Otherwise, search upward in the class hierarchy
- Consequence: the meaning of a message can't be determined statically
- *protocol* of an object: the messages it responds to - determined by its class and superclasses

self and super

- *Not* variables! `self` always refers to the receiver
- `super` always refers to the receiver, but dynamic dispatch is not used; instead:
  - Search upward in the class hierarchy for the method, starting in the superclass *of the class where* `super` *appears in the source*.
  - Result: method is known statically!

The method "new"

- `new` is *not* a keyword - a method in class Class responsible for creating instance variables
- Sometimes we override it, but it's not a good idea to omit "new super":

```
-> (class Bar Object (x)
    (classMethod new ())
    (method x () x))
<class Bar>
-> (val bar (new Bar))
nil
-> (x bar)
run-time error: UndefinedObject does not understand
message x
```


Variable names

- Familiar static scope rules; in order of precedence:
  - locals
  - method parameters
  - instance variables
  - globals

Smalltalk is highly dynamic

- Semantics reflect this
- Almost everything can change at runtime
- (In the full language, even more so!)

Major new features

- Values are objects
  - Object carries its class with it
  - Even classes like SmallInteger can be redefined
  - ... so the behavior of a literal could change during program execution
- Method dispatch - many rules!
- Environments - global and parameter
  - Closures capture *only* parameter environment

Expression evaluation

- Context is a *message send:*
  - global environment ξ
  - local (parameter) environment ρ
  - *static* superclass (superclass of the class where the message send occurs) c_{super}
- Environments map identifiers to locations in the store

Judgments

- Expression evaluation

$$\langle e, \rho, c_{\text{super}}, \xi, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$$

- Top-level evaluation

$$\langle t, \xi, \sigma \rangle \rightarrow \langle \xi', \sigma' \rangle$$

Variables

- Just like Impcore (we ignore the superclass)
- *self* is an instance variable, and *super* behaves like *self* except as the receiver of a message:

$$\frac{\langle \text{VAR}(\texttt{self}), \rho, c_{\text{super}}, \xi, \sigma \rangle \Downarrow \langle v, \sigma \rangle}{\langle \text{SUPER}, \rho, c_{\text{super}}, \xi, \sigma \rangle \Downarrow \langle v, \sigma \rangle}$$

Literals

- Array literals are parsed as VALUES, but numbers and symbols as LITERALS

$$\frac{}{\langle \text{VALUE}(v), \rho, c_{\text{super}}, \xi, \sigma \rangle \Downarrow \langle v, \sigma \rangle}$$

$$\frac{}{\langle \text{LITERAL}(\text{NUM}(n)), \rho, c_{\text{super}}, \xi, \sigma \rangle \Downarrow \langle \langle \sigma(\xi(\texttt{SmallInteger})), \text{NUM}(n) \rangle, \sigma \rangle}$$

Blocks

- We make an object of class *Block*, with a closure as representation, which captures the parameter environment (*not* the globals) as well as the static superclass:

$$\overline{\langle \text{BLOCK}(\langle x_1 \ldots x_n \rangle, es)), \rho, c_s, \xi, \sigma \rangle \Downarrow \langle \langle \sigma(\xi(\texttt{Block})), \text{CLO}(\langle x_1 \ldots x_n \rangle, es, c_s, \rho) \rangle, \sigma \rangle}$$

Message send

- Five cases:
  - user-defined method, receiver is not *super*
  - user-defined method, receiver is *super*
  - primitive method, receiver is not *super*
  - primitive method, receiver is *super*
  - value method

Ordinary user message send

- To evaluate

$$\langle \text{SEND}(m, e, e_1, \ldots, e_n), \rho, c_s, \xi, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$$

- eval receiver and parameters, threading the store:

$$\langle e, \rho, c_s, \xi, \sigma \rangle \Downarrow \langle \langle c, r \rangle, \sigma_0 \rangle$$
$$\langle e_i, \rho, c_s, \xi, \sigma_{i-1} \rangle \Downarrow \langle v_i, \sigma_i \rangle$$

- look up method using receiver's class

$$\texttt{findMethod}(m, c) = \text{USER\_METHOD}(\_, \langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_k \rangle, e_m, \texttt{s})$$

Message send cont'd

- Allocate space for the method's parameters and locals $l_1, \ldots, l_n \notin \text{dom } \sigma_n \quad l'_1, \ldots, l'_k \notin \text{dom } \sigma_n$

$$\hat{\sigma} = \sigma_n\{l_1 \mapsto v_1, \ldots, l_n \mapsto v_n, l'_1 \mapsto nil, \ldots, l'_k \mapsto nil$$

- Create the environment and eval the body

$$\rho' = \texttt{instanceVars}(r)$$

$$\langle e_m, \rho'\{x_1 \mapsto l_1, \ldots, x_n \mapsto l_n, y_1 \mapsto l'_1, \ldots, y_k \mapsto l'_k\}, \texttt{s}, \xi, \hat{\sigma} \rangle \Downarrow \langle v, \sigma' \rangle$$

- Notice which static superclass is used!

Message send to super

- The only difference is that we use the static superclass to start the method lookup.

$\texttt{findMethod}(m, c_s) = \text{USER\_METHOD}(\_, \langle x_1, \ldots, x_n \rangle, \langle y_1, \ldots, y_k \rangle, e_m, \texttt{s})$

Prolog

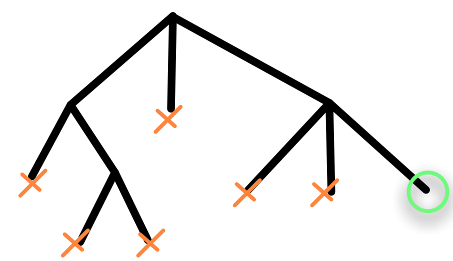
- No evaluation - proof search instead
  - "Variables" are bound as a *result* of search
- A "program" is a set of clauses together with a query
- The meaning of a program is a set of proofs
- The "answer" is yes or no - a proof was found or not - together with bindings for the variables

Unification makes it work

- Unification: given two terms t_1 and t_2, both potentially containing variables, can we find a substitution for those variables making t_1 and t_2 the same?
- e.g. unify `[X,3,4|Xs]` and `[2,3,Y|Ys]`:
  - `{ X:=2, Xs:=Ys, Y:=4 }`

Backtracking makes it work

- A search tree



Logical vs. procedural semantics

- Logical semantics extremely simple but it's an idealization of what actually happens
  - It ignores effects of search order, e.g. nontermination
- Procedural semantics specifies search order
  - Can also specify the behavior of the *nonlogical* constructs like cut

Logical semantics

- Judgment: the conjunction of goals is satisfiable using the set of clauses *D* and the substitution θ

$$D \vdash \hat{\theta} g_1, \ldots, \hat{\theta} g_n$$

- Rule for conjunctions

$$\frac{D \vdash \hat{\theta} g_1 \quad \ldots \quad D \vdash \hat{\theta} g_n}{D \vdash \hat{\theta} g_1, \ldots, \hat{\theta} g_n}$$

Logical semantics cont'd

- Rule for a single goal

$$\frac{\begin{array}{l} C \in D \quad C = G \texttt{:-} H_1, \ldots, H_m \\ \hat{\theta}'(G) = \hat{\theta} g \\ D \vdash \hat{\theta}'(H_1), \ldots, \hat{\theta}'(H_m) \end{array}}{D \vdash \hat{\theta} g}$$

- *C* is *any* clause in the database!

Substitutions

- Informally, think of a substitution as a function that maps logic variables to Prolog terms (which may contain logic variables
- If θ a substitution and t a term, write θt for the application of θ to t
- but write $\hat{\theta}$g for the application to a goal g
- A substitution never affects a functor, predicate, or literal

Unification

- Unification plus variable renaming finds the pair of substitutions we need to match a goal to a clause head
- Why renaming? Consider:

```
member(M,[1|nil])
member(X,[X|M])
```

- We need to consider the two occurrences of M to be different variables.

Unification: two subtleties

- Unification finds a *most general* unifier! We're not interested in other substitutions.
- To be correct, unification must do an *occurs check*: the following should not unify:

```
foo(X,[X|L])
foo(Y,[bar(Y)|M])
```


Procedural semantics

- Specifies order of evaluation
  - which clause is matched first?
  - how does backtracking work?

Choosing a clause

- Given an atomic query *g* and a database *D*, we attempt to satisfy *g* using the clauses of *D* in the order in which they appear.
- This yields nontermination in the following:

```
element(X,[Y|Xs]) :- element(X,Xs).
element(X,[X|Xs]).
?- element(1,L).
```

Backtracking

- If we unify a goal with a clause *C*, but fail to satisfy a subgoal, we return to the list of clauses and try to to unify our goal with the next clause after *C*.
- This causes nontermination in:

```
reach(X,Y) :- reach1(X,Y).
reach(X,Y) :- reach(X,U), reach(U,Y).
reach(X,X).
?- reach(a,a).
```


Comparing the two

- The logical interpretation is "too powerful" - if there is any way to find a proof, it succeeds.
- The procedural interpretation reflects what can be easily, efficiently implemented, but is harder to understand.
- Note that many implementations omit the "occurs check" to speed up unification.

Non-logical features

- Features that cannot be given a simple semantics in pure logic
  - cut (!) prevents backtracking
  - **not** might appear logical but depends on the *closed world assumption*.
  - **assert** and **retract** modify the rules on the fly

Prolog and theorem proving

- Prolog was originally developed as part of research in automated proof
- It is based on *resolution theorem proving,* a proof search procedure for the *Horn clause fragment* of first-order logic