CS301
Session 13

Agenda

- Rules for polymorphic typing

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: $\textsc{tylambda}(\alpha_1,\ldots,\alpha_n,e)$

```
(type-lambda ('a1 ... 'an) exp)
```

- Type application: $\textsc{tyapply}(e,\tau_1,\ldots,\tau_n)$

```
(@ 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

Kinds

- A *kind environment* classifies our types:

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

  and constructors:

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

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

- E.g.,

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

Constructor applications

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

$$\frac{\Delta \vdash \tau :: \kappa_1 \times \ldots \times \kappa_n \Rightarrow \kappa \quad \Delta \vdash \tau_1 :: \kappa_1 \ldots \Delta \vdash \tau_n :: \kappa_n}{\Delta \vdash \textsc{conapp}(\tau, [\tau_1, \ldots, \tau_n]) :: \kappa}$$

- 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:

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

Quantified types

- Where the polymorphism action is:

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

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

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

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

A side excursion

- The word "pair" in the typed uScheme interpreter is heavily overloaded
- "pair" is a type constructor (in the full language)
- "pair" is a polymorphic function that constructs pairs
- "PAIR" is an ML type constructor used to represent values of both list and pair (Scheme) types in the interpreter

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

$$\frac{\Delta, \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n \qquad \Delta, \Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Delta, \Gamma \vdash \textsc{let}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau}$$

- We view let* as syntactic sugar for nested let

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

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

Typing lambda

- We check that the declared parameter types are well formed

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

- 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

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

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!

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

$$\frac{\Delta, \Gamma\{x \mapsto \tau\} \vdash e : \tau}{\Delta, \Gamma \vdash \textsc{val-rec}(x, \tau, e) \rightarrow \Gamma\{x \mapsto \tau\}}$$

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

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

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

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
```


Evaluating VAL and VAL-REC

- Note different handling of environment

$$\frac{l \notin \mathrm{dom}\ \sigma \qquad \langle e, \rho, \sigma\rangle \Downarrow \langle v, \sigma'\rangle}{\langle \textsc{val}(x, e), \rho, \sigma\rangle \rightarrow \langle \rho\{x \mapsto l\}, \sigma'\rangle}$$

- This one's for recursive definitions!

$$\frac{\begin{array}{l} l \notin \mathrm{dom}\ \sigma \\ \langle e, \rho\{x \mapsto l\}, \sigma\{l \mapsto \mathrm{unspecified}\rangle \Downarrow \langle v, \sigma'\rangle \end{array}}{\langle \textsc{val-rec}(x, \tau, e), \rho, \sigma\rangle \rightarrow \langle \rho\{x \mapsto l\}, \sigma'\rangle}$$

CS301
Session 14

Agenda

- A look at where we are
- µML: an introduction

Taking stock

- We've built up typed µScheme from Impcore:
  - Imperative features (assignment, loops, sequencing, output)
  - First-class functions, local bindings
  - Static type checking
  - Polymorphic types

A look ahead

- This week: µML, nearly pure functional programming and type inference
- Two weeks for µSmalltalk: object-oriented programming
- Two weeks for µProlog: logic programming
- One week for ?
  - Programming-in-the-large?
  - Parallel and distributed programming?

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!

Referential transparency on the web

- Google it, but
  - Beware Wikipedja! (Read it, but read the dispute as well if you do)
  - Good: http://foldoc.org/?referential+transparency

Benefits of r.t.

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

A *seriously* r.t. language

- Haskell, which also has lazy evaluation, monads, type classes, and other cool features

Back to 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

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

- μScheme

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

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:

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

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!

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

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 $(\tau_1,...,\tau_n)\tau$
    - note *postfix* notation
  - quantification $\forall\ \tau_1,...,\tau_n.\tau$ 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

Variables

- A variable can have *any* type that's an instance of its type scheme!

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

- E.g. if Γ(*x*) = ∀α.α→α, then *x* can have types int→int, bool→bool, etc.
- This allows for "automatic instantiation" during type inference.

Typing lambda

- Parameter types are under-specified:

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

- Next time we'll see how to find them using unification.

Next time

- Let-polymorphism
- Unification
- A taste of Hindley-Milner type inference

Guide to the reading

- 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
- Look at the *operational interpretation* on p. 265, and try to generate your own for the rules on p. 266
- Try applying some rule to a tiny example