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: TYLAMBDA($\alpha_1, \ldots, \alpha_n, e$)
  (type-lambda ('a1 ... 'an) exp)
- Type application: TYAPPLY($e, \tau_1, \ldots, \tau_n$)
  (@ exp typel ... typen)

Type expressions versus types

- Our language of types is getting fairly complex:
  
  ```plaintext
  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

  \[ \tau_1 \text{ and } \tau_2 \text{ are types} \]

  \[ \frac{}{\tau_1 \to \tau_2 \text{ is a type}} \]

  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:

  \[ \text{int} :: \ast, \text{bool} :: \ast, \text{unit} :: \ast \]

  and constructors:

  \[ \text{list} :: \ast \Rightarrow \ast, \to :: \ast \times \ast \Rightarrow \ast, \text{array} :: \ast \Rightarrow \ldots \]

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

  \[ \text{pair} :: \ast \times \ast \Rightarrow \ast, \text{sum} :: \ast \times \ast \Rightarrow \ast \]

Using kinds

- Kinding rules tell when type expressions are well formed

  \[ \mu \in \text{dom } \Delta \]

  \[ \frac{}{\Delta \vdash \text{TYCON}(\mu) :: \Delta(\mu)} \]

- E.g.,

  \[ \text{list} \in \text{dom } \Delta \]

  \[ \frac{}{\Delta \vdash \text{TYCON}() :: \ast \Rightarrow \ast} \]

Constructor applications

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

  \[ \Delta \vdash \tau :: \kappa_1 \times \ldots \times \kappa_n \Rightarrow \kappa \]

  \[ \Delta \vdash \tau_1 :: \kappa_1 \ldots \Delta \vdash \tau_n :: \kappa_n \]

  \[ \Delta \vdash \text{CONAPP}(\tau, [\tau_1, \ldots, \tau_n]) :: \kappa \]

- 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:

\[ \Delta \vdash \tau_i :: *, \quad 1 \leq i \leq n \]

\[ \Delta \vdash \text{CONAPP}(\text{TYCON}(\text{tuple}, [\tau_1, \ldots, \tau_n])) :: * \]

Quantified types

- Where the polymorphism action is:

\[ \Delta \{\alpha_1 :: *, \ldots, \alpha_n :: *\} \vdash \tau :: * \]

\[ \Delta \vdash \text{FORALL}((\alpha_1, \ldots, \alpha_n), \tau) :: * \]

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

\[ \alpha \in \text{dom} \Delta \]

\[ \Delta \vdash \text{TYVAR}(\alpha) :: \Delta(\alpha) \]

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

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
  \[
  \Delta, \Gamma \vdash e_i : \tau_i, \quad 1 \leq i \leq n
  \]
  \[
  \Delta, \Gamma \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau
  \]
  \[
  \Delta, \Gamma \vdash \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau
  \]
  
- We view let* as syntactic sugar for nested let
  \[
  \Delta, \Gamma \vdash \text{LET}((x_1, e_1), \text{LETSTAR}((x_2, e_2, \ldots, x_n, e_n), e)) : \tau, \quad n > 0
  \]
  \[
  \Delta, \Gamma \vdash \text{LETSTAR}((x_1, e_1, \ldots, x_n, e_n), e) : \tau
  \]
  \[
  \Delta, \Gamma \vdash e : \tau
  \]
  \[
  \Delta, \Gamma \vdash \text{LETSTAR}(\langle \rangle, e) : \tau
  \]

Typing lambda

- We check that the declared parameter types are well formed
  \[
  \Delta \vdash \tau_i :: \ast, 1 \leq i \leq n
  \]
  \[
  \Delta, \Gamma \{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau
  \]
  \[
  \Delta, \Gamma \vdash \text{LAMBDA}(\langle x_1 : \tau_1, \ldots, x_n : \tau_n \rangle, e) : \tau_1 \times \ldots \times \tau_n \rightarrow \tau
  \]
- 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
  \[
  \Delta, \Gamma \vdash e_i : \tau_i, 1 \leq i \leq n
  \]
  \[
  \Delta, \Gamma \vdash e : \tau_1 \times \ldots \times \tau_n \rightarrow \tau
  \]
  \[
  \Delta, \Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \tau
  \]
Typing TYLAMBDA

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

\[
\Delta\{\alpha_1 :: \ast, \ldots, \alpha_n :: \ast\}, \Gamma \vdash e : \tau
\]

\[
\Delta, \Gamma \vdash \text{TYLAMBDA}(\alpha_1, \ldots, \alpha_n, e) : \forall \alpha_1 \ldots \alpha_n. \tau
\]

Typing TYAPPLY

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

\[
\Delta \vdash \tau_i :: \ast, 1 \leq i \leq n
\]

\[
\Delta, \Gamma \vdash e : \forall \alpha_1 \ldots \alpha_n. \tau
\]

\[
\Delta, \Gamma \vdash \text{TYAPPLY}(e, \tau_1, \ldots, \tau_n) : \tau[\alpha_1 \mapsto \tau_1, \ldots, \alpha_n \mapsto \tau_n]
\]

- The resulting type is constructed by \textit{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!

\[
\Delta, \Gamma \vdash e : \tau
\]

\[
\Delta, \Gamma \vdash \text{VAL}(x, e) \rightarrow \Gamma\{x \mapsto \tau\}
\]

\[
\Delta, \Gamma\{x \mapsto \tau\} \vdash e : \tau
\]

\[
\Delta, \Gamma \vdash \text{VAL-REC}(x, \tau, e) \rightarrow \Gamma\{x \mapsto \tau\}
\]

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

\[ \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \]

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

\[ \langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \]

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

VAL - a pitfall

- Suppose VAL doesn't always create a new binding

\[
\begin{align*}
\text{> uscheme} \\
\to (\text{val x 1}) \\
1 \\
\to (\text{define f (n) (+ x n)}) \\
f \\
\to (f 2) \\
3 \\
\to (\text{val x '(a b)}) \\
(a b) \\
\to (f 2) \\
\text{error: in (+ x n), expected an integer, but got (a b)}
\end{align*}
\]

VAL pitfall (2

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

\[
\begin{align*}
\text{> tuscheme} \\
\to (\text{val x 1}) \\
1 : \text{int} \\
\to (\text{define int f ((int n)) (+ x n)}) \\
f : (\text{function (int) int}) \\
\to (f 2) \\
3 : \text{int} \\
\to (\text{val x '(a b)}) \\
(a b) : (\text{list sym}) \\
\to (f 2) \\
3 : \text{int}
\end{align*}
\]

Evaluating VAL and VAL-REC

- Note different handling of environment

\[
\begin{align*}
l \not\in \text{dom } \sigma \\
\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle \\
\langle \text{VAL}(x, e), \rho, \sigma \rangle \to \langle \rho\{x \mapsto l\}, \sigma' \rangle
\end{align*}
\]

- This one’s for recursive definitions!

\[
\begin{align*}
l \not\in \text{dom } \sigma \\
\langle e, \rho\{x \mapsto l\}, \sigma\{l \mapsto \text{unspecified}\} \rangle \Downarrow \langle v, \sigma' \rangle \\
\langle \text{VAL-REC}(x, \tau, e), \rho, \sigma \rangle \to \langle \rho\{x \mapsto l\}, \sigma' \rangle
\end{align*}
\]
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 Wikipedia! (Read it, but read the dispute as well if you do)
  - Good: [http://foldoc.org/?referential+transparency](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
  \[
  \begin{array}{ll}
  \langle e_1, \rho \rangle & \Downarrow v_1 \\
  \langle e_2, \rho \rangle & \Downarrow v_2 \\
  \vdots \\
  \langle e_n, \rho \rangle & \Downarrow v_n \\
  \end{array}
  \]

- µScheme
  \[
  \begin{array}{ll}
  \langle e_1, \rho, \sigma_0 \rangle & \Downarrow \langle v_1, \sigma_1 \rangle \\
  \langle e_2, \rho, \sigma_1 \rangle & \Downarrow \langle v_2, \sigma_2 \rangle \\
  \vdots \\
  \langle e_n, \rho, \sigma_{n-1} \rangle & \Downarrow \langle v_n, \sigma_n \rangle \\
  \end{array}
  \]

\[
\begin{align*}
\langle \text{BEGIN}(e_1, e_2, \ldots, e_n), \rho, \sigma \rangle & \Downarrow \langle v_n, \sigma \rangle \\
\langle \text{BEGIN}(), \rho \rangle & \Downarrow \langle \text{bool}(\#f), \sigma \rangle
\end{align*}
\]
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:

\[
\langle e, \rho \rangle \Downarrow \langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e_c), \rho_c \rangle
\]

\[
\langle e_1, \rho \rangle \Downarrow v_1 \ldots \langle e_n, \rho \rangle \Downarrow v_n
\]

\[
\langle e_c, \rho_c \{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \} \rangle \Downarrow v
\]

\[
\langle \text{APPLY}(e, e_1, \ldots, e_n), \rho \rangle \Downarrow v
\]

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!

\[
e_1, \ldots, e_n \text{ are all LAMBDA expressions}
\]

\[
\rho' = \rho\{ x_1 \mapsto v_1, \ldots, x_n \mapsto v_n \}
\]

\[
\langle e_1, \rho' \rangle \Downarrow v_1 \ldots \langle e_n, \rho' \rangle \Downarrow v_n
\]

\[
\langle e, \rho' \rangle \Downarrow v
\]

\[
\langle \text{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e), \rho \rangle \Downarrow v
\]

Recursion (3)

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

```ml
datatype value = NIL
    | CLOSURE of lambda * (unit -> value env)

fun eval(e, rho) = let fun ...
    | ev(LETX (LETREC, bs, body)) = ...
    let fun makeRho' () = ...
      in  eval(body, makeRho'())
      end
    end
end
```

- Shallow embedding again!
Recursion for lambda only!

- In µScheme:

```scheme
(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)))
```

- In µML:

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

Type system

- Once again we have type expressions of
  - variables \(\alpha\)
  - constructors \(\mu\)
  - 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:

```scheme
-> (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 allowed in µML:

```scheme
-> (val too-poly (lambda (nil)
   (pair (cons 1 nil) (cons #t nil))))

type error: Cannot unify int and bool
```

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!

\[ \Gamma(x) = \sigma \quad \tau <: \sigma \]

\[ \Gamma \vdash x : \tau \]

- E.g. if \( \Gamma(x) = \forall \alpha. \alpha \rightarrow \alpha \), then \( x \) can have types int\( \rightarrow \)int, bool\( \rightarrow \)bool, etc.

- This allows for "automatic instantiation" during type inference.

Typing lambda

- Parameter types are under-specified:

\[ \Gamma(x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n) \vdash e : \tau \]

\[ \Gamma \vdash \text{lambda}(\langle x_1, \ldots, x_n \rangle, e) : \tau_1 \times \ldots \times \tau_n \rightarrow \tau \]

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

Next time

- Let-polymorphism
- Unification
- A taste of Hindley-Milner 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.

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.