DPL-8: ML and Type Inference

Types, type schemes and type environments

We build types - as in μScheme - using four elements:
• Type variables - denoted by α
• Type constructors - generally denoted by τ or using a specific name such as int or list.
• Constructor application - for which we use the ML notation (t₁, ..., tₙ)τ
• Quantification, which we write using ∀

Difference: in μML types are more restricted than in μScheme

- A "clear" advantage? - at least from the point of view of writing code
- "Formal replacement" would correspond to μML types
- A "formal replacement" would correspond to μML types

ML Examples

<table>
<thead>
<tr>
<th>ML Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>val id = fn x =&gt; x</td>
</tr>
<tr>
<td>becomes (in μScheme)</td>
</tr>
<tr>
<td>&lt;vol id (type-lambda ('a) (lambda (('a x)) x)))</td>
</tr>
<tr>
<td>3:() becomes (in μScheme)</td>
</tr>
<tr>
<td>((# cons int) 3 (@ '() int))</td>
</tr>
</tbody>
</table>

A "clear" advantage? - at least from the point of view of writing code...

Instance Relation:

i' < α = ∀α₁,...,αₙ. i' = i a substitution i' such that i' = i α and ∀i' ∈ (α₁, ..., αₙ), iα' = α'. The substitution i can substitute only for variables that appear under ∀.

To instantiate a type scheme, we choose a i' < α.

- Can be viewed as a) a function from type variables to type variables;
- a function from types to types; c) a function from type schemes to type schemes;
- a function from type environments to type environments.

Representing Hindley-Milner types - a bit more detail

The key notation is τ < α: instantiation τ is an instance of α.

- The types to substitute are unspecified

Representing Hindley-Milner types - shallow types

Ex.: assume we have a type scheme ∀α.(α → α), and we want to instantiate it with a type scheme for α such that ∀(i. i → int)

A "formal replacement" would correspond to ∀α.(α → α) ∀(i. i → int) = ∀(i. i → int) ∀(i. i → int)

If we were to interpret the resulting object ∀(i. i → int) ∀(i. i → int) as a type, there is no way we could write it as ∀(i. ?) with any type expression ?.

The only possible resulting type would be ∀(i. i → int) ∀(i. i → int)

The first "result" has an arbitrary choice of i on both sides of the main →, the second has the same i.
DPL-8: ML and Type Inference

Hindley-Milner: Key ideas elaborated
The type environment $\Gamma$ binds a variable to a type scheme
- We frequently have the degenerate case with zero universally quantified variables (a monotype)
The judgment $\Gamma \vdash e : \tau$ gives the expression a type
At use, we automatically instantiate a type scheme
At let binding, we automatically abstract over types
- This is the so-called "Milner's Let" (Milner's LET)
- A refinement: abstract only over type variables not bound in the environment
In typed Scheme the instantiation and abstraction were explicit.

DPL-8: ML and Type Inference

Key ideas formalized: some ML type (checking) rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma(x) = \alpha$</td>
<td>$\Gamma \vdash x : \alpha$ (Var)</td>
</tr>
<tr>
<td>$\Gamma \vdash e : \tau$</td>
<td>$\Gamma \vdash \text{let } x \in \alpha \text{ in } e : \tau$ (let)</td>
</tr>
<tr>
<td>$\Gamma \vdash \text{Apply}(e_1, e_2) : \tau$</td>
<td>$\Gamma \vdash \lambda(x_1, \ldots, x_n) : \tau$ (LAMBDA)</td>
</tr>
</tbody>
</table>

ML type rules, continued

<table>
<thead>
<tr>
<th>Rule</th>
<th>Statement</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Gamma \vdash \lambda(x_1, \ldots, x_n) : \tau$</td>
<td>$\Gamma \vdash \lambda(x_1, \ldots, x_n) : \tau$ (LAMBDA)</td>
</tr>
<tr>
<td>$\Gamma \vdash \text{MLET}(e_1, e_2, e) : \tau$</td>
<td>$\Gamma \vdash \text{MLET}(e_1, e_2, e) : \tau$ (MLET)</td>
</tr>
</tbody>
</table>

The notion of generalization
Given a type $\tau$ and a type environment $\Gamma$, $\tau = \forall \alpha \; \Gamma \vdash e : \tau$, where $\{\alpha_1, \ldots, \alpha_n\} = \text{fv}(\tau) \cap \text{fv}(\Gamma)$

$\Gamma \vdash e : \tau$ | $\Gamma \vdash \forall \alpha \; e : \tau$ (MLET) |

The notion of instantiation (another pass)
Given a type $\tau$ and a type environment $\Gamma$, $\lambda(x_1, \ldots, x_n) \vdash e : \tau$, where $\{x_1, \ldots, x_n\} = \text{fv}(\tau) \cap \text{fv}(\Gamma)$

Actions: remove the quantifiers of $\epsilon$ and substitute types (in a consistent way) for previously quantified variables and then add quantifiers for any variables in the new expression, except when the variable already occurred free (unquantified) in the original type scheme $\epsilon$.

$\forall \alpha \; (\beta \to \alpha) \vdash \forall \alpha \; (\beta \to \alpha)$

but not

$\forall \alpha \; (\beta \to \alpha) \vdash \forall \alpha \; (\beta \to \alpha)$

since $\beta$ occurs free in $\forall \alpha \; (\beta \to \alpha)$. 

Materials Copyright N. Ramsey & al 11/7/06
**DPL-8: ML and Type Inference**

**Things to notice**

- Lambda-bound variables are monomorphic
- Let-bound variables are polymorphic
- We have to guess the right types!

**Examples:**

```plaintext
- let fun id x = x in (id 3, id false) end;
  val it = (3, false) : int * bool
- fun test tt x y = (tt x, tt y); val ('a, 'b) test = fn ('a -> 'b) -> 'a -> 'a -> 'b * 'b
- test (fn x => x) 3 false;
```

**Type clash:** bool cannot be int...

**Type inference**

- also known as type reconstruction

**How can we make it work?**

- We need to have a type variable for each unknown type
- As information becomes available, we substitute appropriate constructors for the type variables
- The accumulated information is kept as a set of substitutions
- The substitution is driven by unification

**Instances and substitution - more formal pass**

**Examples:**

```plaintext
int <: 'a
int list <: 'a list
but not int <: 'a list
```

And the instance relation \( \tau_1 \prec \tau_2 \): (finally made precise)

- \( \tau_i \) is an instance of \( \tau_2 \) (\( \tau_2 \) is more general than \( \tau_i \))
- if and only if \( \exists \theta : \tau_1 = \theta(\tau_2) \), where \( \theta \) is a transformation.

We use substitutions (i.e. the results of transformations) in type inference.

**Instance properties**

**Theorem:** \( 'a \) is the most general type

(every type \( \tau \) is an instance of \( 'a \))

**Proof:** let \( \theta = (a \mapsto t) \)

**Theorem:** \( < \) is a partial order.

**Proof:**
1) reflexive: \( \lambda \tau . \tau \), the identity substitution gives \( \tau = \theta(\tau) \) for all \( \tau \).
2) transitive: \( \tau_1 = \theta(\tau_2) , \tau_2 = \theta(\tau_3) \Rightarrow \tau_1 = \theta(\tau_3) \).
3) antisymmetric: since \( \lambda \tau . \tau \) is the only transformation which is its own inverse.

**Instantiation intuition**

Consider an application \((e_1, e_2)\)

- \( e_1 \) must have some arrow type, call it \( \alpha \mapsto \beta \)
- \( e_2 \) must have some type, call it \( \gamma \)
- \( e_2 \) must have some type, call it \( \gamma \)
- The result type is instantiated \( \beta \)
- We want the most general instantiation

**Example:** length [true, false]

- The type of \( e_1 \) is \( 'a list \mapsto \) int
- The type of \( e_2 \) is \( bool \) list
- Instantiate with \( \theta = ('a \mapsto bool) \); the result type is int
To make the idea precise we say that \( \tau_1 \) and \( \tau_2 \) are unified by the transformation \( \theta \) if
\[
\theta \tau_1 = \theta \tau_2
\]

Ex: 'a * int and int * 'b are unified by the transformation
\[
('a' \rightarrow \text{int}) o (\text{int}' \rightarrow 'b)
\]

The substitution \( \theta \) represents assumptions about type variables.

Most General Unifiers

A most general unifier will make no unnecessary assumptions about type. Note that:

- 'a list and 'b are unified by
  \[
  ('a' \rightarrow \text{int list}) o ('b' \rightarrow \text{int list})
  \]
- 'a list and 'b are unified by
  \[
  ('b' \rightarrow 'a list)
  \]

Which unifier is more general?

Def.: A transformation \( \theta_1 \) is an instance of a transformation \( \theta_2 \) if there exist a transformation \( \theta \) such that
\[
\theta_1 = \theta \circ \theta_2
\]

Def.: A most general unifier of types \( \tau_1 \) and \( \tau_2 \) is a substitution \( \theta \) s.t.

- \( \tau_1 \) and \( \tau_2 \) are unified by
  \[
  \theta \tau_1 = \theta \tau_2
  \]
- there is no more general unifier \( \theta \) that unifies \( \tau_1 \) and \( \tau_2 \)

Implementing unification

\( \text{unify}(\tau_1, \tau_2) \) returns the most general unifier.

Recall types as
\[
\tau : \rightarrow \text{TYVAR} \alpha | \text{TYCON} \mu | \text{CONAPP}(\tau_1, [\tau_1, \ldots, \tau_n])
\]

TYVAR \( \alpha \) unifies with any type \( \tau \) by \( \alpha \tau \), provided \( \alpha \) is not free in \( \tau \) (or \( \alpha = \tau \)) - ex.: we can't unify 'a with 'a list, and \( x :: x \) must not have a type.

TYCON \( \mu \) unifies only with itself (\( \text{int} \) unifies only with \( \text{int} \))

CONAPP unifies with CONAPP if the arguments unify

The outermost function: calls \( \text{internal-unify} \) and decides what to return in case of failure or success.

\[
\begin{align*}
\text{unify-match}(p_1, p_2, f) & \rightarrow \text{the-empty-stream} \quad \text{if } \text{eq? result \( \text{false} \)}}
\end{align*}
\]

\[
\begin{align*}
\text{the-empty-stream} & \rightarrow \text{singleton-stream result})
\end{align*}
\]

\( p_1 \) and \( p_2 \) are patterns; \( f \) is an initial variable transformation (usually empty)
DPL-8: ML and Type Inference

```lisp
;;; internal-unify (THE GUYS)
(define (internal-unify p1 p2 frame)
  (cond ((eqv? p1 (if (constant? p2) 'failed) 'failed))
        (if (constant? p1) 'failed)
        ((constant? p2) 'failed)
        ((eqv? p1 p2) (if (unique? p1) 'failed)
                       (internal-unify (car p1) (cdr p2)
                            (internal-unify (car p2)
                                             (car p1)
                                             frame))))
```

DPL-8: ML and Type Inference

```lisp
;;; internal-unify (THE GUYS)
(define (internal-unify p1 p2 frame)
  (cond ((eqv? p1 (if (constant? p2) 'failed) 'failed))
        (if (constant? p1) 'failed)
        ((constant? p2) 'failed)
        ((eqv? p1 p2) (if (unique? p1) 'failed)
                       (internal-unify (car p1) (cdr p2)
                            (internal-unify (car p2)
                                             (car p1)
                                             frame))))
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
```

DPL-8: ML and Type Inference

```
DPL-8: ML and Type Inference
From type rules to type inference, II

\[
\Gamma \vdash \text{LAMBDA}(\{x_1, \ldots, x_n\} : \tau_1 \times \cdots \times \tau_n \rightarrow \tau) \quad \text{(LAMBDA)}
\]
becomes

\[
\theta(\Gamma \vdash \text{LAMBDA}(\{x_1, \ldots, x_n\} : \theta_{\tau_1} \times \cdots \times \theta_{\tau_n} \rightarrow \theta_{\tau})) \quad \text{(LAMBDA)}
\]

NB. \(\theta(\Gamma) \vdash \text{LAMBDA}(\{x_1, \ldots, x_n\} : \theta_{\tau_1} \times \cdots \times \theta_{\tau_n} \rightarrow \theta_{\tau})\)

Materials Copyright N. Ramsey & al.

11/7/06

DPL-8: ML and Type Inference
From type rules to type inference, III

\[
\Gamma \vdash e_i : \tau_i \rightarrow \tau \quad \Gamma \vdash e_i : \tau_i \rightarrow \tau \quad \text{(APPLY)}
\]
becomes

\[
\theta(\Gamma) \vdash e_i : \theta_{\tau_i} \rightarrow \theta_{\tau} \quad \text{(APPLY)}
\]

Materials Copyright N. Ramsey & al.

11/7/06

DPL-8: ML and Type Inference
More explicit substitutions

\[
\Gamma \vdash e : \tau \quad \{a_1, \ldots, a_n\} \vdash \text{fv}(e) = \text{fv}(\Gamma)
\]

\[
\Gamma \vdash \text{MLET}(x_1, x_2 : \tau) \quad \text{(MILLER'S LET)}
\]

becomes

\[
\theta(\Gamma) \vdash e : \theta_{\tau} \quad \{\theta(a_1, \ldots, a_n)\} \vdash \text{fv}(\theta(e)) = \text{fv}(\theta(\Gamma))
\]

\[
\theta(\theta(\Gamma)) \vdash \text{MLET}(x_1, x_2 : \theta_{\tau}) \quad \text{(MILLER'S LET)}
\]

Materials Copyright N. Ramsey & al.

11/7/06

DPL-8: ML and Type Inference
Type inference, operationally

Like type checking
\- top-down, bottom-up pass over abstract syntax
\- use \(\Gamma\) to look up types of variables
Different from type checking
\- get substitution \(\theta\)
\- use \(\theta\) to modify \(\Gamma\)
\(\Gamma\) represents assumptions

Materials Copyright N. Ramsey & al.

11/7/06

DPL-8: ML and Type Inference
Operational example

Start with \(\text{val}\) \(f = \text{fn} x \rightarrow \)
Add binding \(x \rightarrow \theta\) to \(\Gamma\)
no assumptions about \(x\) (it's an unknown type)
continue with body: \(\text{val}\) \(f = \text{fn} x \rightarrow x + 1\)
Look up \(+\) in \(\Gamma\), find \(\theta : \text{int} \times \text{int} \rightarrow \text{int}\)
Unify \(\text{ty} x, \text{ty} 1\) : \(\theta : \text{int} \times \text{int} \rightarrow \text{int}\)
The most general unifier is \(\theta : \theta'\)
We modify the environment: \(\theta'(x \rightarrow \theta)\)
The most general unifier is \(\theta : \theta' \rightarrow \text{int}\)
we'll give you a suitable abstract data type...

Materials Copyright N. Ramsey & al.

11/7/06

6
DPL-8: ML and Type Inference

### Implementing type inference

Calling `typeof(e, Gamma)` returns a pair `(tau, theta)` such that theta on Gamma _α_ : tau

$$\theta(A) = \theta(x_i \rightarrow \Gamma_i)$$

where _α_ is fresh

$$(\exists \theta_\alpha :: \Gamma \rightarrow \theta \rightarrow \alpha)$$

(Amst)

Write `funty = t->lu, actualtypes = v_1, ..., v_n, rettype = \alpha`

Fun type `funtype : (Gamma, actuals)` =

- let val `(funty :: actualtypes, theta) = (* cheat *)`
- `typeof : (Gamma, actualtypes)` =
- `val rettype = freshtype`
- `val theta = unify(funtypes, actualtypes, rettype)`

in `(theta, rettype, theta' & theta)` end

DPL-8: ML and Type Inference

### More Unification and Substitution: generalization

$$\Gamma \vdash \alpha$$

$$(\exists \theta_\alpha :: \Gamma \rightarrow \theta \rightarrow \alpha)$$

$$\Gamma \vdash \theta \alpha$$

where _α_ is fresh

$$\theta \alpha \Rightarrow \theta \alpha$$

$$\theta \alpha \Rightarrow \theta \alpha$$

$$\Gamma \vdash \theta \alpha$$

$$\theta \alpha \Rightarrow \theta \alpha$$

$$\Gamma \vdash \theta \alpha$$

### An inference example, III

1. `(define (exists? p? lis)...)`
2. `(if (null? lis) #f (exists? (cdr lis)))`...
3. Initial type environment \(\Gamma\) - instantiate forall
4. `null`: `c list -> bool`
5. `#: bool`
6. `#: bool`
7. `if`: `bool * 'e * 'e -> 'e`
8. `car`: `f list -> 'f`
9. `cdr`: `g list -> 'g list`

### An inference example, IV

- `(null? lis)` implies `i -> 'c list`
- `(p? (car lis))` implies `j -> 'c -> 'l`
- `(if (...) # (exists? ..))` implies `k -> 'n -> bool`
- `(exists? p? ..)` implies `m -> ('c -> bool) * 'n`
- `(exists? .. (cdr lis))` implies `n -> 'c list`

Therefore, composing all substitutions yields:

- `lis`: `'c list`
- `p?`: `'c -> bool`
- `exists?`: `('c -> bool) * ('c list -> bool`

DPL-8: ML and Type Inference

### Real ML - type inference

- ML has no assignment.
- let `val r = ref (fn x => x)`
- `in r = (fn x => x + 1); 1` (true); end
- `would calculate (true = 1). Must not be allowed to type check!`
- `r has type ('a -> 'a)` ref

**Problem:** typing rules use substitution of any term, evaluation only substitutes values.

**Fix:** the value restriction = the right-hand side of a let declaration is polymorphic only if it is a value.
DPL-8: ML and Type Inference

Real ML: type ascriptions
ML allows the user to state a type for an expression. The type given must be an instance of the type inferred for the expression. The ascribed type will then be used as the type of the expression (see what MoscowML does with memofun).

```ml
val ei : int list = []
val memofun : (int -> int) ref = 
  fn _ => raise Fail "Not initialized";
fun f x = memofun (x - 1) _
val _ = memofun := memoize f
```

What is not Hindley-Milner typeable?
Consider the Church numerals, with type:

\[ \forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha) \to (\alpha \to \alpha) \]

`suc` takes a Church numeral and returns a Church numeral:

\[ (\forall \alpha. (\alpha \to \alpha) \to (\alpha \to \alpha)) \to (\forall \beta. (\beta \to \beta) \to (\beta \to \beta)) = \forall \beta. \exists \alpha. ((\alpha \to \alpha) \to (\alpha \to \alpha)) \to ((\beta \to \beta) \to (\beta \to \beta)) \]

Try `add`, `mult`, `exp` at type.

```
val expt = fn m => fn n => fn f => fn x => ((( n m) f) x);
val ('a,'b,'c,'d) expt = fn : ('a -> ('a -> 'b -> 'c -> 'd)) -> 'b -> 'c -> 'd
val expt1 = expt : (('a -> 'a) -> 'a -> 'a) -> (('a -> 'a) -> 'a -> 'a) -> ('a -> 'a) -> 'a -> 'a;
```

Warning: Value polymorphism:
Free type variable(s) at top level in value identifier it

```
val it = fn : ('a -> 'a) -> 'a -> 'a
```

Type systems: things to remember.
Compile-time checking
Type soundness
Type erasure
Examples
1. Simple monomorphic: like C
2. General polymorphic: super-powerful, but requires too much notation
3. Hindley-Milner polymorphic: powerful, no notation
Some runnable terms are not typeable.

The µML Interpreter: abstract syntax and values.
The concrete syntax looks much like that of µScheme - with some minor exceptions (val rec ...).
The abstract syntax:
```
datatype exp = LITERAL of value |
| VAR of name |
| IIX of exp * exp * exp |
| BEGIN of exp list |
| LETX of let_kind * (name * exp) list * exp |
| LAMBDA of name list * exp |
| APPLY of exp * exp list |
and let_kind = LET | LETREC | LETSTAR
```
DPL-8: ML and Type Inference

The μML Interpreter: abstract syntax and values.

and value = NIL | BOOL of bool | NUM of int | SYM of name | PAIR of value * value | CLOSURE of lambda * (unit -> value \(\text{en}\)) | PRIMITIVE of \(\text{primop}\) withtype \(\text{primop} = \text{value list} \rightarrow \text{value}\) (* raises RuntimeException*) and lambda = name list * exp exception RuntimeException of string (* error message *)

At toplevel we have the objects
datatype toplevel = EXP | DEFINE of name * (name list * exp) | VAL of name * exp | VALREC of name * exp | USE of name

Note the BEGIN two slides back: sequencing construct, since this language contains two imperative primitives: error and print.

DPL-8: ML and Type Inference

The μML Interpreter: Operational Semantics.

This is quite simple, and we provide just one example - see the text for a fairly complete set of semantic judgments.

\[
\begin{align*}
&\text{IF} e_1, e_2, e_3(\text{)}, \#v_2
&\text{IF TRUE(\text{}\text{)}}
&\text{e}_1, \#v_1 \text{= BOOL#f(\text{}\text{)}}\ e_3, \#v_3
&\text{IF} e_1, e_2, e_3(\text{}\text{)}, \#v_3
&\text{IF FALSE(\text{}\text{)}}
\end{align*}
\]

DPL-8: ML and Type Inference

The μML Interpreter: Type Systems.

Recall that we build types using four elements:

- Type variables - denoted by \(\alpha\)
- Type constructors - generally denoted by \(\mu\) or using a specific name such as \(\text{int}\) or \(\text{list}\).
- Constructor application - for which we use the ML notation \((\text{\tau}_1, \ldots, \text{\tau}_n)\text{\tau}\)
- Quantification, which we write using \(\forall\) - the symbol \(\sigma\) stands for a quantified type:

\[\sigma ::= \forall \alpha_1, \ldots, \alpha_n. \tau\]

DPL-8: ML and Type Inference

The μML Interpreter: Type Systems.

We need to formalize the notion of substitution and then write code.

1. function from type variables to type variables
2. function from types to types
3. function from type schemes (quantified) to type schemes
4. function from type environments to type environments

fun tysubst (tau, varenv) = let fun subst (TYVAR a) = (find(a, varenv) handle NotFound => TYVAR a) | subst (TYCON c) = (TYCON c) | subst (CONAPP (tau, taus)) = CONAPP (subst tau, map subst taus) in subst tau end

NOTE: in Scheme we needed \(\text{kinds}\) - to prevent the programmer from introducing malformed types. Since μML does not allow the programmer to introduce types, there is no need for \(\text{kinds}\).
DPL-8: ML and Type Inference

The µML Interpreter: Type Systems.

Instantiation requires us to find the appropriate instance of a type scheme; quantification will disappear from the result - we use only variables that appear under ∀.

```ml
fun instantiate (FORALL (formals, tau), actuals) =
  tysubst(tau, bindList (formals, actuals, emptyEnv))
handle BindListLength =>
  let exception ThisCan'tHappen
  in raise ThisCan'tHappen
  end
```

There are some other functions (p. 260) to manipulate types (i.e. construct type expressions).

DPL-8: ML and Type Inference

The µML Interpreter

There are some other functions (p. 260) to manipulate types (i.e. construct type expressions).

```ml
fun freevars t =
  let fun f(TYVAR v,          l) = insert (v, l)
        | f(TYCON _,          l) = l
        | f(CONAPP(
          ty, tys),  l) = foldl f (f (ty, l)) tys
  in  rev (f(t, emptyset))
  end
```

Recall that a number of the type judgments required the introduction of a fresh type variable: ML itself provides us with a mechanism that hides the counter - a similar mechanism exists in Scheme…

```ml
local
val n = ref 1
in
  fun freshtyvar_ =
    TYVAR ('t' ^ Int.toString (!n) before n := !n + 1)
end
```

DPL-8: ML and Type Inference

The µML Interpreter: Type Environments

Function `bindtype`: adds binding `x : σ` to `Γ` (for LAMBDA and LET).

```ml
fun bindtype (v, sigma as FORALL (bound, tau), (Gamma, theta, free)) =
  (bind(v, sigma, Gamma), theta, union(diff (freevars tau, bound), free))
```

Function `findtype`: look up a variable `x` in `Γ`, finds `Γ(x) = σ`, then uses fresh type variables to return a most general instance `τ <: σ` (for VAR).

```ml
fun findtype (v, (Gamma, theta, free)) = theta (freshInstance (find(v, Gamma)))
```

Function `on`: applies a substitution `θ` to an environment `Γ`. May not substitute for bound variables.

```ml
infixr 2 on
fun theta on (Gamma, oldtheta, free) =
  let val free = foldl union emptyset (map (freevars o theta o TYVAR) free)
  in (Gamma, theta o oldtheta, free)
  end
```

DPL-8: ML and Type Inference

The µML Interpreter: Type Environments.

Function `on`: applies a substitution `θ` to an environment `Γ`. May not substitute for bound variables.

```ml
function on : applies a substitution θ to an environment Γ. May not substitute for bound variables.
```

```ml
fun literal _ = raise LeftAsExercise
```

```ml
fun ty (LITERAL n) = literal n
```

```ml
| ty (VAR v) = (findtype (v, Gamma), idsubst)
```

DPL-8: ML and Type Inference

The µML Interpreter: Type Environments.

The main function is

```ml
fun typeof (e, Gamma) =
  let fun typesof ([], Gamma : type_env) = ([], idsubst)
        | typesof (e::es, Gamma) =
            let val (tau, theta)  = typeof (e, Gamma)
                val (taus, theta') = typesof (es, theta on Gamma)
            in  (theta' tau :: taus, theta' o theta)
            end
  in  typesof (e::es, Gamma)
  end
```

```ml
function typeof : applies a substitution θ to an environment Γ. May not substitute for bound variables.
```
The µML Interpreter: Type Inference.

At top level we have

fun topCheckEval (t, envs as (Gamma, rho), echo) =
  case t of
  USE filename => use readCheckEvalPrint filename envs
  | _ =>
    let
      val (Gamma, tystring) = topty (t, Gamma) (* get type *)
      val (rho, valstring) = topeval (t, rho) (* get value *)
      val _ = if size valstring > 0 then echo (valstring ^ " : " ^ tystring) else ()
    in
      (Gamma, rho)
    end

There are several places where unify is called: the first order of business is to implement it. Then extend the type inference system to the types that are left as an exercise.

The µML Interpreter: Type Inference.

fun topCheckEval (t, envs as (Gamma, rho), echo) =
  case t of
  USE filename => raise RuntimeError "internal error -- `use' reached top" 
  | EXP e =
    let
      val (tau, _) = typeof (e, Gamma)
      val sigma = generalize (tau, Gamma)
    in
      (Gamma, typeSchemeString sigma)
    end
  | DEFINE (x, lambda) =>
    topty (VALREC (x, LAMBDA lambda), Gamma) 
  | VAL (x, e) =
    let
      val (tau, _) = typeof (e, Gamma)
      val sigma = generalize (tau, Gamma)
    in
      (bindtype (x, sigma, Gamma), typeSchemeString sigma)
    end
  | VALREC (x, e) =
    let
      val tau = freshtyvar ()
      val Gamma' = bindtype (x, FORALL ([], tau), Gamma)
      val (tau', theta) = typeof (e, Gamma')
      val theta' = unify (tau', theta)
      val sigma = generalize (theta', tau')
    in
      (bindtype (x, sigma, Gamma), typeSchemeString sigma)
    end