Static Semantics: Types in ML

The next two lectures introduce more details of the ML type system. After that we will look at type analysis and inference in ML, i.e. at static semantic analysis. For this we will introduce ML_0, a much simplified form of ML and we will discuss type calculus based on ML_0.

Polymorphic Functions

Consider the identity function:

```
- fun id x = x;
 val id = fn : 'a -> 'a
```

This function works on all types of arguments: reals, lists, functions, etc.

It is not overloaded, it is polymorphic. Its type

```
'a -> 'a
```

is a type scheme, and `'a` **is a type variable.**

Some more examples:

```
 fun len []      = 0
   | len (x::xs) = 1 + len xs;
 val len = fn : 'a list -> int

 fun reverse []    = []
   |  reverse (x::xs) = (reverse xs) @ [x];
 val reverse = fn : 'a list -> 'a list

 fun fst (x, y) = x;
 val fst = fn : 'a * 'b -> 'a

- fst (("abc", 7), ("def", 6));
 val it = ("abc",7) : string * int
- fst (3.0,1);
 val it = 3.0 : real
```

Polymorphism

Polymorphic type checking is a secure yet flexible type discipline. Most ML programs need not be cluttered with type specifications, as types are deduced automatically.

ML is strongly typed:

"Well-typed programs cannot go wrong."

Once the type checker has accepted the program, no type errors can occur at run-time.

(Division by zero is not a "type error!")

A polymorphic function can have different types within the same expression:

```
- len [1.0,2.0] + len ["abc","def"]
 val it = 4 : int
```

Equality Types

There is a slight problem with polymorphism and equality.

```
- fun mem(x,[])     = false
   |  mem(x,y::ys) = (x = y) orelse mem(x,ys);
 val mem = fn : ''a * ''a list -> bool
```

If `''a` is a real (or a function type), `mem` will want to compare.

```
- mem(3,[1,2,3]);
 val it = true : bool
- mem(3.0,[1.0,2.0,3.0]);
 stdIn:140.1-140.23 Error: operator and operand
             don't agree [equality type required]
   operator domain: ''Z * ''Z list
   operand:         real * real list
   in expression:
     mem (3.0,1.0 :: 2.0 :: <exp> :: <exp>)
```

ML will correctly not allow this since reals are not comaparable in ML (neither are function types).

Equality Types (Cont.)

The built-in function = is polymorphic `''a * ''a -> bool`**.**

We don't need to write a special function to test list equality:

```
- [2,3,4] = [2,3,4];
 val it = true : bool
```

The types admitting equality testing are called equality types. Type variables ranging over these are `''a`**,** `''b`**, etc.**

Equality testing is possible for most types, including tuples and lists made from equality types.

Equality Types (Cont.)

What is the difference between these three functions? Why?

```
fun len1 []      = 0
  | len1 (x::xs) = 1 + len1 xs;
```

val len1 = fn : 'a list -> int

```
fun len2 x = if x=[] then 0 else 1 + len2(tl x);
```

val len2 = fn : ''a list -> int

```
fun len3 x = if null x then 0 else 1 + len3(tl x);
```

val len3 = fn : 'a list -> int

Basic Types (Review)

Types in ML (as in most typed languages) are defined recursively with a basis of primitive types and then rules for defining more complex types from these.

Basic types: `int`, `real`, `char`, `bool`, `unit`, `string`

Product type: `'a * 'b`, **more generally** `'a * 'b * 'c *....`

Function type: `'a -> 'b`.

Type constructors: We have met `list`, **ie** `'a list`.

In this lecture we shall see how to define your own type constructor.

Type Definitions

We can define a new type in terms of an existing type. It acts as an abbreviation (renaming). Their general form is:

```
type <identifier> = <type expression>.
```

Much like a `typedef` **in C. For instance:**

```
type ints = int list;
```

Also, recall our functions for manipulating complex numbers:

```
type complex = real * real;
fun addComplex ((x1,y1),(x2,y2)) =
     (x1+x2,y1+y2): complex;
fun subComplex ((x1,y1),(x2,y2)) =
     (x1-x2,y1-y2): complex;
fun multComplex ((x1,y1),(x2,y2)) =
     (x1*x2 - y1*y2, x1*y2 + x2*y1): complex;
```

```
type complex = real * real
val addComplex = fn :
     (real * real) * (real * real) -> complex
val subComplex = fn :
     (real * real) * (real * real) -> complex
val multComplex = fn :
     (real * real) * (real * real) -> complex
val it = () : unit
```

Note that ML recognizes that `complex` **and** `real * real` **are the same.**

Parameterized Type Definitions

More generally, we can pass type variables as parameters to the definition.

```
type (<list of type parameters>) <identifier> =
 <type expression>.
```

Imagine an association list which has a domain type (the key) and a range type (the value). It "maps" domain elements to range elements, and implements a dictionary.

```
- type ('d, 'r) assoclist = ('d * 'r) list;
 type ('a,'b) assoclist = ('a * 'b) list

- val phonenumbers = [("Peter",56790345),
      ("Nicole",56790345)]:
      (string,int) assoclist;
 val phonenumbers = [("Peter",56790345),
      ("Nicole",56790345)]:
      (string,int) assoclist;
```

The `<identifier>` **in the type definition identifies a type constructor (of a given arity – the number of its parameters).**

The `<type expression>` **will be built from type constructors and from type variables appearing in** `<list of type parameters>`

Datatypes

ML has a powerful mechanism for defining new types called datatypes. Their general form is:

```
datatype (<list of type parameters>) <identifier> =
                  <first constructor expression> |
                  <second constructor expression> |
                   ...
                  <last constructor expression>
```

The `<identifier>` **again identifies a (data)type constructor.**

Each `<constructor expression>` **is built from a data constructor whose arguments (if any) are type variables appearing in** `<list of type parameters>` **and type constructors.**

For example:

```
- datatype fruit = Apple | Pear | Grape;
 datatype fruit = Apple | Grape | Pear
```

Here `fruit` **is the new data type and** `Apple`**,** `Pear`**,** `Grape` **are data constructors with no arguments.**

Datatypes (Cont.)

```
- fun isApple x = (x=Apple);
 val isApple = fn : fruit -> bool

- isApple(Apple);
 val it = true : bool
- isApple(Pear);
 val it = false : bool
- isApple(Banana);
 stdIn:21.9-21.15 Error:
   unbound variable or constructor: Banana
```

Intutively, a variable of type `fruit` **is just a tag indicating which kind of data constructor it is.**
(Like an `enum` **in C++).**

However, data constructors can be much more powerful...

Tagged Union Datatype Example

Data constructors can have arguments. The constructor does not affect the argument, it simply adds a tag to the argument.

The tag allows arguments constructed in different ways to be distinguished by pattern matching.

```
type name = string;
type id = int;
type degree = string;

datatype student =
       Bachelor of name*id*degree |
       PhD of name*id |
       Master of name*id*degree;

fun name (Bachelor(n,_,_)) = n
  |   name (PhD(n,_)) = n
  |   name (Master(n,_,_)) = n;

 val name = fn : student -> name
```

Note that the extra brackets are needed if you use pattern matching on data constructors that have arguments.

Also note that while type constructors build types, data constructors build values of a type.

Falafel Rolls

Data constructors can be recursive.

A falafel roll has pita bread as its base and might have the following ingredients

- **falafel balls**
- **tabouli**
- **pickles**
- **hummus**
- **chilli**

We can use the following datatype to model falafel rolls

```
datatype falafelRoll =
  Pita |
  Falafel of falafelRoll |
  Tabouli of falafelRoll |
  Pickles of falafelRoll |
  Hommus of falafelRoll |
  Chilli  of falafelRoll;

- val yummy = Chilli(Falafel(Tabouli(Falafel(Pita))));

 val yummy =
   Chilli (Falafel (Tabouli (Falafel (Pita)))):falafelRoll
```

Falafel Rolls

A real falafel roll is one with some falafel balls in it:

```
datatype falafelRoll =
      Pita |
      Falafel of falafelRoll |
      Tabouli of falafelRoll |
      Pickles of falafelRoll |
      Hommus of falafelRoll |
      Chilli  of falafelRoll;

fun realFalafel Pita = false
|    realFalafel (Falafel(r)) = true
|    realFalafel (Tabouli(r)) = realFalafel(r)
|    realFalafel (Pickles(r)) = realFalafel(r)
|    realFalafel (Hommus(r)) = realFalafel(r)
|    realFalafel (Chilli(r)) = realFalafel(r);

 val realFalafel = fn : falafelRoll -> bool

- realFalafel yummy;
 val it = true : bool
```

Complex Datatypes

Generally, in datatype definitions:

- **Type variables can be used to parametrize the definition.**
- **The data constructors can take arguments.**
- **They may be recursive.**

Recall the general form:

```
  datatype (<list of type parameters>) <identifier> =
    <first constructor expression>   |
    <second constructor expression>  |
...
    <last constructor expression>
```

List Datatype

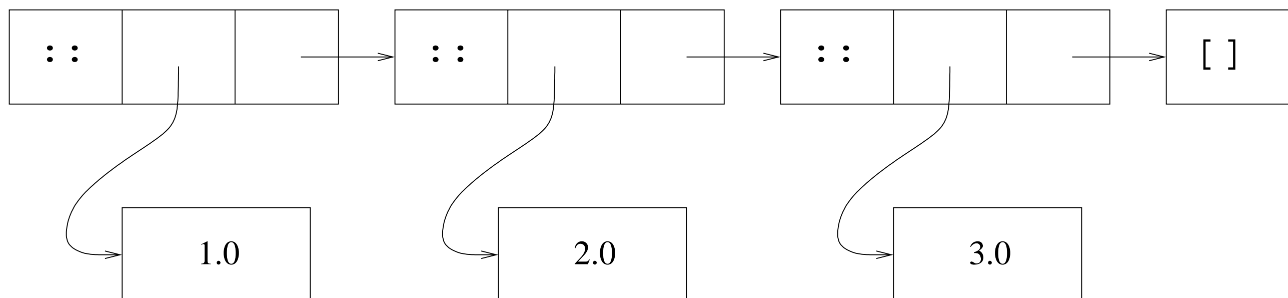
For example the `list` data type is conceptually defined by

```
datatype 'a list =
  [] |
  :: of 'a * 'a list;
```

The list `[1.0,2.0,3.0]` which is shorthand for

```
1.0 :: 2.0 :: 3.0 :: []
```

is represented by



Do not to confuse data constructors with functions:
– data constructors build data
– functions compute things from the data.

Binary Trees

The following defines a “labelled” binary tree:

```
datatype 'label btree =
  Empty |
  Node of 'label btree * 'label * 'label btree;

datatype 'a btree = Empty |
    Node of 'a btree * 'a * 'a btree

- val names = Node(Empty,"Kim",Empty);
 val names = Node (Empty,"Kim",Empty) : string btree
```

Records

A record is like a tuple, but its components (fields) are named and situated between curly brackets.

The records

```
{name = "Jones", age = 25, height = 180}
{height = 180, name = "Jones", age = 25}
```

are equal.

```
val jones_info =
  {name = "Jones",
   age = 25,
   height = 180
  };

val jones_info = {age=25,height=180,name="Jones"}
  : {age:int, height:int, name:string}
```

Selection of a field is done using #$\langle label \rangle$**.**

```
- #age jones_info;
val it = 25 : int
- #height(jones_info);
val it = 180 : int
```

Tuples

Actually tuples are really a record whose fields are called `#1`, `#2, ...`

Thus:

```
- #2 (3.6, "Select Me", 6);
 val it = "Select Me" : string
```

Records (Cont.)

We can pick fields and assign by matching:

```
- val {age = theage, height = theheight,...} = jones_info;
 val theage = 25 : int
 val theheight = 180 : int
```

Or we can use the field name as the variable itself:

```
- val {name, age,...} = jones_info;
 val age = 25 : int
 val name = "Jones" : string
```

The `...` **is part of the ML syntax—it means "the rest of the record." (like a wild card)**

Records (Cont.)

We can give the record type a name and let functions have arguments and/or results of the type:

```
 type info = {name : string,
     age : int,
     height : int
     };
  type info = {age:int, height:int, name:string}

- fun silly ({age, height,...}) = height-age;
 stdIn:55.1-55.39 Error:
  unresolved flex record (need to know the names
  of ALL the fields in this context)
   type: {age:'Y, height:'Y; 'Z}
- fun silly ({age, height,...} : info) = height-age;
 val silly = fn : info -> int
- silly jones_info;
 val it = 155 : int
```

Functions as Expressions

Recall that we could write `len` **as**

```
val rec len = fn
      []      => 0
    | (x::xs) => 1 + len xs;
```

Notice that `fn P1 => E1 | P2 => E2 | ... | Pk => Ek` **is an expression – it is an anonymous function.**

As far as ML is concerned function definitions are just expressions and so, like any other expression they do not need a name.

```
- (fn x => x+1)(3);
 val it = 4:int

- fn x => x+1;
 val it = fn : int -> int
- it 3;
 val it = 4 : int
```

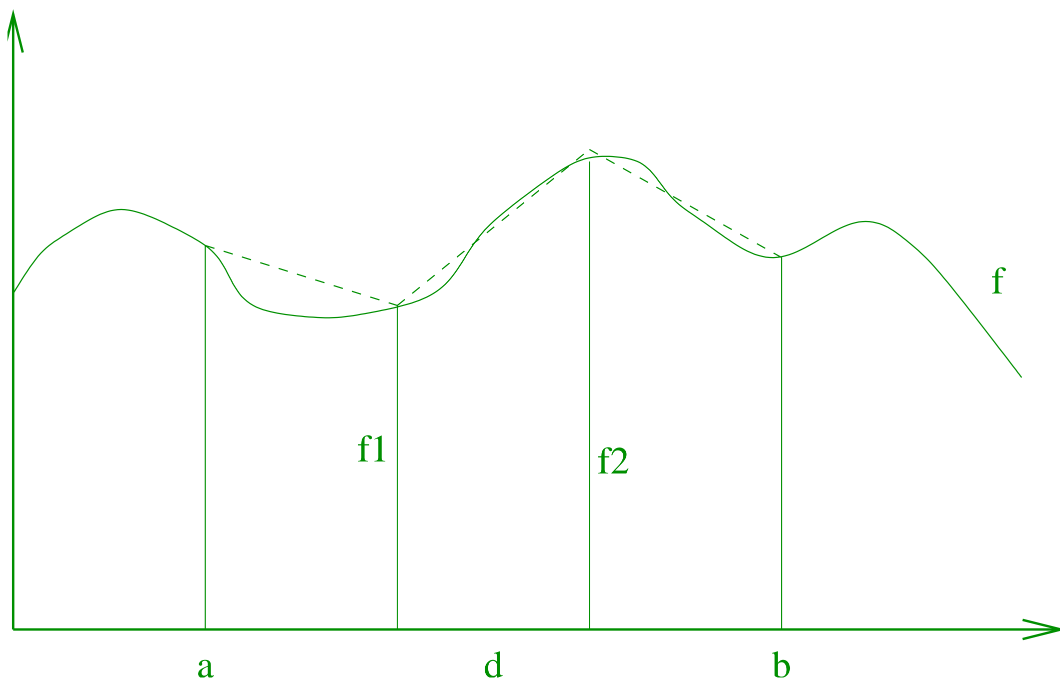
Higher Order Functions

Like any other expression functions can be used as arguments to functions.

Functions that take functions as arguments are said to be higher-order.

Higher-order programming is easy to do in ML.

One way to compute the integral of a function is to use the trapezoidal rule:



The area of the ith trapezoid is

$$\delta \times \frac{f(a + (i-1) \times \delta) + f(a + i \times \delta)}{2}$$

where $\delta = \frac{b-a}{n}$**.**

Higher Order Functions

```
(* function for integration using the trapezoid rule*)
fun trap(a,b,n,F) =
  if n<=0 orelse b-a <= 0.0 then 0.0
  else
    let
       val delta = (b-a)/real(n)
    in
       delta * (F(a)+F(a+delta))/2.0 +
       trap(a+delta,b,n-1,F)
  end;

 fun square x = x*x : real;
```

Use this to compute $\int_0^1 x^2 dx$**.**

```
- use "trap.ml";
 [opening trap.ml]
 val trap = fn:real * real * int * (real -> real) -> real
 val square = fn : real -> real
 val it = () : unit

- trap(0.0, 1.0, 10, square);
 val it = 0.335 : real

- trap(0.0, 1.0, 10, (fn x => x*x));
 val it = 0.335 : real
```

Another Example

In one of the earlier lectures you were asked to write a polymorphic function to find the largest item in a list.

```
(* finds the maximum element in a list *)
exception EmptyList;
fun max [] = raise EmptyList
  | max [x] = x
  | max (x::xs) =
     let val xsmax = max xs in
       if x > xsmax then x else xsmax
     end;
```

val max = fn : int list -> int

```
- max [4,6,3,2,6,8];
```

val it = 8 : int

This is not possible since > is not polymorphic.

Another Example (Cont.)

However we can write a function which takes the comparison function gt **as an argument.**

```
fun max (gt, []) = raise EmptyList
  | max (gt, [x]) = x
  | max (gt, (x::xs)) =
     let val xsmax = max (gt, xs) in
       if gt (x,xsmax) then x else xsmax
     end;

 val max = fn : ('a * 'a -> bool) * 'a list -> 'a

- val igt = fn (x:int,y) => x>y;
 val igt = fn : int * int -> bool
- val sgt = fn (x:string,y)  => x>y;
 val sgt = fn : string * string -> bool
- max (igt,[4,6,3,2,6,8]);
 val it = 8 : int
- max (sgt,["abc","def","ghi"]);
 val it = "ghi" : string
```

The Simple Map Function

The simple map function takes a function F and a list $[a_1, ..., a_n]$ and returns the list $[F(a_1), ..., F(a_n)]$.

(Similar to `mapcar`).

```
fun simpleMap(F,[]) = []
 |  simpleMap(F,x::xs) =
       F(x)::simpleMap(F,xs);

 val simpleMap = fn : ('a -> 'b) * 'a list -> 'b list

- simpleMap(square,[1.0,4.0,3.0]);
 val it = [1.0,16.0,9.0] : real list

- simpleMap(~,[1,2,3]);
 val it = [~1,~2,~3] : int list
```

The Simple Map Function (Cont.)

How does `simpleMap(~,[1,2])` **execute?**

Reduce

The reduce function takes a binary function F and a list $[a_1, ..., a_n]$ and returns

$$F(a_1, F(a_2, F(..., F(a_{n-1}, a_n)))).$$

```
exception EmptyList;

fun reduce(F,[]) = raise EmptyList
 |  reduce(F,[a]) = a
 |  reduce(F,x::xs) = F(x,reduce(F,xs));

fun plus(x,y) = x+y;

 exception EmptyList
 val reduce = fn : ('a * 'a -> 'a) * 'a list -> 'a
 val plus = fn : int * int -> int

- reduce(plus,[3,4,7,10]);
 val it = 24 : int

- reduce(+,[3,4,7,10]);
 stdIn:39.8 Error: expression or pattern begins with
                   infix identifier "+"

- reduce(op +,[3,4,7,10]);
 val it = 24 : int

- reduce(fn (x,y)=>x+y, [3,4,7,10]);
 val it = 24 : int
```

The Reduce Function (Cont.)

How does `reduce(plus,[3,4])` **execute?**

Filter

The filter function takes a Boolean function P and a list $[a_1, ..., a_n]$ and returns the sublist whose elments satisfy P.

```
fun filter(P,[]) = []
 |  filter(P,x::xs) =
      if P(x) then x::filter(P,xs)
      else filter(P,xs);
```

*val filter = fn : ('a -> bool) * 'a list -> 'a list*

```
- filter(fn(x)=> x>10, [1,10,23,45,8]);
```

val it = [23,45] : int list

Example

Exercise: **Consider a function** `concat` **which takes a list of strings and concatenates them all.**

```
- concat ["abc","def","ghi"];
 val it = "abcdefghi" : string
```

Write a version which is recursive and write another which uses `reduce`**.**

Returning a Function

Since functions are just like any other expression a function can also return a function!

Such functions are also said to be higher-order.

For example we can have a function which takes a number x and returns a function to add x on to its argument:

```
fun add x = (fn y => x+y);
```

val add = fn : int -> int -> int

Or if x is positive adds it and otherwise subtracts it:

```
fun strange x = if x > 0 then (fn y => x+y)
                    else (fn y => y-x) ;
```

val strange = fn : int -> int -> int

Returning a Function (Cont.)

How do we use these?

```
- add 3;
 val it = fn : int -> int
- it 4;
 val it = 7 : int
- (add 3) 4;
 val it = 7 : int
```

Since function application is considered left-associative we can even leave out the parentheses —

```
- add 3 4;
 val it = 7 : int
```

Currying of Functions

Consider a function with a pair as argument:

```
- fun add' (x,y) = x + y : int;
 val add' = fn : int * int -> int
```

But `add x y = add' (x,y)`**!**

What is the difference?

`add'` **requires both** `x` **and** `y` **before it can be evaluated while** `add` **only requires** `x`**, the other "argument"** `y` **can be given later.**
Thus `add` **is more flexible.**

`add` **is the curried version of** `add'`**.**
(Named after the logician Haskell Curry.)

It is often possible to avoid tuples as arguments to functions, through currying.

Currying of Functions (Cont.)

ML makes it easy to define curried functions.

We can define `add` **by**

```
fun add x y = x+y;
 val add' = fn : int -> int -> int
```

This is just shorthand for

```
fun add x = (fn y => x+y);
```

Which is just shorthand for

```
val add = (fn x => (fn y => x+y));
```

"Real" ML programmers generally use curried functions rather than tuples.

Realistic Example: Binary Search Trees

```
datatype 'label btree =
  Empty |
  Node of 'label btree * 'label * 'label btree;

fun lookup lt Empty x = false
 |  lookup lt (Node(left,lbl,right)) x =
      if lt(x,lbl) then (lookup lt left x)
      else if lt(lbl,x) then (lookup lt right x)
      else (* x=lbl *) true;

fun insert lt Empty x = Node(Empty,x,Empty)
 |  insert lt (T as Node(left,lbl,right)) x =
      if lt(x,lbl) then Node((insert lt left x),lbl,right)
      else if lt(lbl,x) then Node(left,lbl,(insert lt right x
      else (* x=lbl *) T; (* already in tree *)
```

```
 val lookup = fn :
       ('a * 'a -> bool) -> 'a btree -> 'a -> bool
 val insert = fn :
       ('a * 'a -> bool) -> 'a btree -> 'a -> 'a btree
```

Binary Search Trees (Cont.)

```
- val tree = Node(Empty,"harald",Node(Empty,"peter",Empty));
 val tree =
 Node (Empty,"harald",Node (Empty,"peter",Empty)) : string t
- val tree = insert (op <) tree "karen";
 val tree =
 Node (Empty,"harald",Node (Node #,"peter",Empty)) : string
- lookup  (op <) tree "karen";
 val it = true : bool
- lookup  (op <) tree "carine";
 val it = false : bool
```

Built-In Higher Order Functions

Function Composition. Recall

$$(G \circ F)(x) = G(F(x)).$$

We can define this by

```
fun comp F G x = G(F(x));
 val comp = fn : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c

- val mystery = comp square square;
 val mystery = fn : real -> real

- mystery 5.0;
 val it = 625.0 : real
```

ML provides the infix operator "o" to compose two functions.

```
- val mystery = square o square;
 val mystery = fn : real -> real
```

Built-In Higher Order Functions (Cont)

Map. ML provides a curried version of map.

```
- map;
 val it = fn : ('a -> 'b) -> 'a list -> 'b list

- val mystery = map square;
 val mystery = fn : real list -> real list

- mystery [1.0,16.0,9.0];
 val it = [1.0,256.0,81.0] : real list
```

Foldr and Foldl. More powerful curried versions of reduce. The definition of `foldr` is:

```
fun foldr F y [] = y
 |  foldr F y (x::xs) = F(x, foldr F y xs)


- val sumList = foldr op + 0;
 val sumList = fn : int list -> int
- sumList [2,4,7];
 val it = 13 : int
```

Summary

We have looked at:

- More Details of the ML type system
  - Polymorphism
  - Datatype Definitions
  - Tuples and Record
- Higher-order functions

Homework

- Study the following Sections of Ullman's ML book: Sec. 5.3,–5.4, Sec. Sec. 6, Sec. 7.1.

Type Inference for ML-like Languages

In this lecture we will look at type handling **in semantic analysis**

- Type inference / reconstruction for ML-like languages
- Hindley-Milner type system

Related Material, on which this presentation is loosely based, can be found in Wilhelm and Maurer, Chapter 9 (particularly Section 9.1) and in Carl A. Gunter, "Semantics of Programming Languages", MIT Press, Cambridge/MA 1992, Chapter 7 (particularly Section 7.5) as well as in the material for the current version of the "Programming Languages" subject at Harvard University.

Reminder: Semantic Analysis

The semantic analysis determines non-syntactic properties of the program as well as properties that cannot (easily) be determined by a context-free grammar.

Type checking **and** type inference **are among the most important tasks of the semantic analysis.**

For example, recall that $L = \{a^n b^m c^n d^m \mid \; n, m \geq 1\}$ **is not context-free.**

Therefore it is not a context-free property whether or not the number of parameters in function declarations and in their calls agree.

Checking whether even the types agree is a much harder problem!

Reminder: Type Checking with Attribute Grammars

In some cases the types of expressions can be determined by a bottom up (post-order) tree traversal of the syntax tree.

As attribute grammars are more powerful than "pure" context-free grammars, this can in principle be embedded in an attribute grammar.

Recall the example grammar for simple assignment expressions:

Grammar rule: $assgn \rightarrow ident := exp$
Attribution Rules:

```
assgn.operation :=
  if ident.type = int then int_asg else real_asg
ident.env := assgn.env
exp.env := assgn.env
```

Condition:

```
coercible(exp.type,ident.type) and
ident.kind = var
```

Grammar rule: $exp1 \rightarrow exp2 + exp3$
Attribution Rules:

```
exp1.type :=
  if exp2.type = int and exp3.type = int
  then int else real
exp1.operation :=
  if exp1.type = int then int_add else real_add
exp2.env := exp1.env
exp3.env := exp1.env
```

Condition:

```
coercible(exp2.type,exp1.type) and
coercible(exp3.type,exp1.type)
```

Grammar rule: *exp* → *ident*
Attribution Rules:

```
exp.type :=
  lookup(ident.symbol, exp.env)
```

Type Checking versus Type Inference

Note how in the above grammar the type of the operations is decided from the types of the operands: for a simple expression the information on the types flows in principle only in one direction (as long as the environment is only used for lookup).

Such a computation is only possible if the types of all identifiers are declared explicitly and all type coercions are explicit.

Consider the ML example:

```
fun  q (x::xs) y = 1::(q xs y) |
     q [] y = y ;

> val q = fn : 'a list -> int list -> int list
```

To determine (implicit) types information needs to be propagated in a more complex way.

Hindley-Milner Type Inference

The Hindley-Milner type system is the basis for languages like ML, Objective Caml, Haskell etc.

It is a formal method for type inference in ML-like languages and offers a restricted form of parametric polymorphism (let-polymorphism).

- **Type abstraction happens at binding time**
- **Type application happens when a function is applied**

We introduce a language ML_0 to discuss type inference. It represents the core of ML-like languages.

$$
\begin{array}{lcl}
expr & \rightarrow & var \\
expr & \rightarrow & fn\ var_1 \ldots var_n => expr \\
expr & \rightarrow & expr(expr_1 \ldots expr_n) \\
expr & \rightarrow & if\ expr\ then\ expr\ else\ expr \\
expr & \rightarrow & let\ var\ =\ expr\ in\ expr \\
var & \rightarrow & x \mid\ \ldots \mid z \mid\ \ldots
\end{array}
$$

Note that all expressions are implicitly typed.

Type inference in implicitly typed languages is also called type reconstruction.

Type Schemas

To represent (polymorphic) types we use type schemas. We have

- **type variables** $'a$, . . .
- **simple types** $bool, int, \ldots$
- **polymorphic types** $'a\ list, \ldots$ **(by application of type constructors)**

A grammar for type schemas:

$$\begin{array}{lcl} type & \rightarrow & typevar \mid\ bool \mid \ldots \mid\ int \\ type & \rightarrow & typeconstructor(type_1, \ldots, type_n) \\ schema & \rightarrow & type \mid\ \forall x.type \end{array}$$

These are the normal ML types, for example:

$int, bool, real, 'a\ , 'b\ , \mapsto (int, \mapsto (int, real)), \mapsto (list('a\ ), list('a\ ))$.

Note

- **for technical convenience we use an alternative notation for type constructors, i.e. we write them like function applications. Example:** $list(int, real) = (int * real)\ list$
- **types like** $list('a\ )$ **are implicitly quantified (**$'a$ **is quantified over all possible types),**

We also need to introduce the idea of an instantiation:

$'a\ <: 'b$ **means that** $'a$ **is an instantiation of (more specific than) the type schema** $'b$ **, for example** $int\ list <: \forall 'a\ .'a\ list$.

Judgement

We also need the idea of type environments and judgements.

- **A type environment Γ contains all the type bindings**
- **A judgement on an expression e, written $\Gamma \vdash e : t$ gives the expression e the type t in environment Γ.**

Γ is a collection of type bindings $x : t$.

The judgement is the most fundamental type inference justified by the rule:

$$H, x : t, H' \vdash x : t$$

This means that we can make the judgement $\Gamma \vdash e : t$ if the environment contains the binding t for the type variable x.

Type Rules

Next we need to introduce the notion of how to build up more and more complex type judgements. For this we use type rules, i.e. inference rules for types.

We write type rules (inference rules for types) in the following form:

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : {'a} \quad \Gamma \vdash e_3 : {'a}}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : {'a}} \; (If)$$

This can be read in either of two ways:

- **bottom-up: To show that "if e_1 then e_2 else e_3" is an expression of type $'a$ we need to show that e_1 is of type $bool$ and e_2, e_3 are both of type $'a$.**
- **top-down: If we know that e_1 is of type $bool$ and e_2, e_3 are both of type $'a$ we may conclude that "if e_1 then e_2 else e_3" is an expression of type $'a$.**

ML_0 **Type Rules**

$$\frac{\Gamma = H, x : T, H' \quad 'a \ <: T}{\Gamma \vdash x : 'a} \ (Var)$$

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : 'a \quad \Gamma \vdash e_3 : 'a}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : 'a} \ (If)$$

$$\frac{\Gamma \vdash e_i : 'a_{\ i}, i = 1 \ldots n \quad \Gamma \vdash f : 'a_{\ 1} \rightarrow \ldots \rightarrow 'a_{\ n} \rightarrow 'a}{\Gamma \vdash (f \ e_1 \ldots e_n) : 'a} \ (Apply)$$

$$\frac{\Gamma, x_1 : 'a_{\ 1}, \ldots, x_n : 'a_{\ n} \vdash e : 'a}{\Gamma \vdash (\text{fn } x_1 \ldots x_n => \ e) : ('a_{\ 1} \rightarrow \ldots \rightarrow 'a_{\ n} \rightarrow 'a\ )} \ (Lambda)$$

$$\frac{\begin{array}{c}\Gamma \vdash \hat{e} : '\hat{a} \\ \Gamma, (x : \forall 'a_1\ , \ldots, 'a_{\ n}.'\hat{a}\ ) \vdash e : 'a \\ \{'a_1\ , \ldots, 'a_{\ n}\} = fv('\hat{a}\ ) - fv(\Gamma)\end{array}}{\Gamma \vdash (let\ x \ = \ \hat{e}\ in\ e) : 'a} \ (Let)$$

Note:

- **The name "Lambda" for function abstraction stems from the roots of functional languages in the Λ-calculus.**
- $x_i \mapsto 'a_{\ i}$ **in (Lambda) is implicitly assumed to be a universal closure:** $x_i \mapsto \forall 'a_{\ i}$**.**

Type Inference

To perform type inference we have to do the following:

- Represent the type of each expression with unknown type by a fresh variable
- Collect these types in a type environment
- using the type rules, decompose the expression stepwise
  - apply a type rule
  - substitute type variables in the environment according to the type information in the rule

  until the expression is completely decomposed and all types are known.

The key notion here is the substitution. It means that we replace a type variable with a type schema (i.e. a concrete type, a type variable or a polymorphic type).

Substitutions

By substituting a type variable we can make a type schema only more specific. A type schema in which a type variable has been substituted is an instance of the schema without substitution, for example:

$$\begin{aligned} &int <: {}'a \\ &int\ list <: {}'a \\ &int\ list <: {}'a\ list \end{aligned}$$

We write $\{T/'a\}$ to indicate that we substitute $'a$ with T. To apply a substitution Θ to an expression e we write Θe, for example

$\Theta = \{int/'a, int\ list/'b\}$
$e = {}'a \times {}'a \mapsto {}'b$
$\Theta e = int \times int \mapsto int\ list$**.**

We can express the instance relation $'a <: {}'b$ by substitution:

$'a <: {}'b$ **if and only if** $\exists \Theta . {}'a = \Theta {}'b$

Finding the right substitutions

When applying a type rule perform the following steps.

Example: Infer the type of function application $e_1(e_2)$

$$\frac{\Gamma \vdash e_i : 'a_{\ i}, i = 1 \ldots n \quad \Gamma \vdash f : 'a_{\ 1} \rightarrow \ldots \rightarrow 'a_{\ n} \rightarrow 'a}{\Gamma \vdash f(e_1, \ldots, e_n) : 'a} \ (Apply)$$

- **denote the types of all expressions by fresh type variables** $\Gamma = e_1 : 'b\ , e_2 : 'c$ **.**
- **the rule application dictates that we must have** $\Theta' b \ = \ 'b_{\ 1} \mapsto 'b_{\ 2}$**,** $\Theta' a_{\ 1} = \Theta' b_{\ 1}$**,** $\Theta' a \ = \Theta' b_{\ 2}$**.**

Substitution Example

```
val f = fn (x, y) => y+1
val g = fn y => f (0, y)
```

What is the type of `f(0, y)`**?**

Clearly we have: `f: ‘a * int -> int` **and** `g: int -> int`

With `(0,y): int * ‘b` **we obtain**

$\Theta = \{int/'a\ , int/'b\ \}$.

When deriving the type of an expression we are only allowed to make substitutions that are as general as possible, example:

```
- val f = fn (x,y) => (x,y)::[];
> val f = fn : ’a * ’b -> (’a * ’b) list
- val g = fn x => f(1,x);
> val g = fn : ’a -> (int * ’a) list
```

Note that $'a$ **remains unsubstituted.**

Unification

Finding the most general instantiation is done via unification of type schemata.

Two type schemata are unified by a substitution Θ if

$\Theta' a \ = \Theta' b$ **.**

Θ **is called the unifier**

We need to find the most general unifier (MGU).

A unifier Θ_1 is more general than another unifier Θ_2 if,

$$\exists \Theta . \Theta \circ \Theta_1 = \Theta_2$$

i.e. there is another substitution Θ that makes Θ_1 equal to Θ_2.

Example: $\Theta_1 = \{'b \ list/'a \ \}$, $\Theta_2 = \{int \ list/'a \ \}$, $\Theta = \{int/'b \ \}$

Θ_2 **is more general than** Θ_1**.**

A most general unifier for $'a$ and $'b$ is a substitution Θ such

- $\Theta('a \ ) = \Theta('b \ )$
- $\neg \exists \Theta' . \Theta'('a \ ) = \Theta'('b \ )$ **and Θ' is more general than Θ.**

Extending Type Rules with Unification (If)

We now rewrite the type rules from above to do the following:

From a given type environment Γ and expression e we compute the most general substitution Θ such that e is well typed in $\Theta\Gamma$.

From

$$\frac{\Gamma \vdash e_1 : bool \quad \Gamma \vdash e_2 : {}'a \quad \Gamma \vdash e_3 : {}'a}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : {}'a} \; (If)$$

we obtain

$$\frac{\Theta\Gamma \vdash e_1, e_2, e_3 : {}'a_{\ 1}, {}'a_{\ 2}, {}'a_{\ 3} \quad \Theta'({}'a_{\ 1}) = bool \quad \Theta'({}'a_{\ 2}) = \Theta'({}'a_{\ 3})}{(\Theta' \circ \Theta)\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \Theta'({}'a_3\ )}$$

Extending Type Rules with Unification (Lambda)

From

$$\frac{\Gamma, x_1 : {'a}_1, \ldots, x_n : {'a}_n \vdash e : {'a}}{\Gamma \vdash (\text{fn } x_1 \ldots x_n \text{=> } e) : ({'a}_1 \to \ldots \to {'a}_n \to {'a})} \; (Lambda)$$

we obtain

$$\frac{\Theta(\Gamma, x_1 : {'a}_1, \ldots, x_n : {'a}_n) \vdash e : {'a}}{\Theta\Gamma \vdash (\text{fn } x_1 \ldots x_n \text{=> } e) : (\Theta({'a}_1) \to \ldots \to \Theta({'a}_n) \to {'a})}$$

Extending Type Rules with Unification (Apply)

From

$$\frac{\Gamma \vdash e_i : {'a}_{\ i}, i = 1 \ldots n \quad \rightarrow \vdash f : {'a}_{\ 1} \rightarrow \ldots \rightarrow {'a}_{\ n} \mapsto {'a}}{\Gamma \vdash (f \ \ e_1 \ldots e_n) : {'a}} \ (Apply)$$

we obtain

$$\frac{\begin{array}{c}\Theta\Gamma \vdash f, e_1, \ldots, e_n : {'a} \ , {'a}_{\ 1} \ldots {'a}_{\ n} \\ \Theta'({'a} \ ) = \Theta'({'a}_{\ 1} \rightarrow \ldots \rightarrow {'a}_{\ n} \rightarrow {'b} \ ) \text{ where } {'}\mathrm{b} \text{ is a fresh type variable}\end{array}}{(\Theta' \circ \Theta)\Gamma \vdash (f \ \ e_1 \ldots e_n) : \Theta'({'b} \ )}$$

Extending Type Rules with Unification (Let)

From

$$\frac{\begin{array}{c}\Gamma \vdash \hat{e} : {'\hat{a}} \\ \Gamma, x : \forall {'a_1}, \ldots, {'a_n}.{'\hat{a}} \vdash e : {'a} \\ \{{'a_1}, \ldots, {'a_n}\} = fv({'\hat{a}}) - fv(\Gamma)\end{array}}{\Gamma \vdash (let\ x\ =\ \hat{e}\ in\ e) : {'a}} \ (Let)$$

we obtain

$$\frac{\begin{array}{c}\Theta'\Gamma \vdash \hat{e} : {'\hat{a}} \\ \Theta((\Theta'\Gamma), x : \forall {'a_1}, \ldots, {'a_n}.{'a'}) \vdash e : {'a} \\ \{{'a_1}, \ldots, {'a_n}\} = fv({'\hat{a}}) - fv(\Theta'\Gamma)\end{array}}{(\Theta \circ \Theta')\Gamma \vdash (let\ x\ =\ \hat{e}\ in\ e) : {'a}}$$

Note: $fv(e)$ **are the free variables in expression** e**.**

How to Perform Type Reconstruction

On the basis of these type rules and unification we can perform type inference (reconstruction) for ML like languages in much the same way as for fully (explicitly typed) expressions.

Perform a pass over the syntax tree

- **type each unknown expression with a fresh type variable**
- **add the fresh variable to the type environment**
- **get the current substitution Γ**
- **find the MGU Θ on the basis of the type rule to be applied**
- **modify the environment to be $\Theta\Gamma$**

Type Reconstruction Example

To reconstruct the type of

```
val f = fn x => x + 1
```

- **start with head** `fn x`
- **add new type binding** $x : {}'a$ **to** Γ**, nothing is yet known about this type**
- **continue with the body** `x + 1`
- **lookup definition of** $+$**.**
  $\Gamma \vdash + : int \times int \mapsto int$**.**
- **Unify** `(x,1): ('a * int)` **with** `int*int`**.** $\Theta = \{int/'a\ \}$**.**
- **modify the environment** Γ **by applying** Θ**. The new current environment in** $\Theta\Gamma$**.**

Another Example

Reconsider the introductory example (modified to use `fn` instead of `fun`).

```
val rec q = fn ([],y)      => y
             | (x::xs, y) => (1::(q (xs,y))) ;

> val q = fn : 'a list * int list -> int list
```

initialize the type environment:

$1 : int, q : 'a\ , x : 'b\ , xs : 'c\ , y : 'd$

`q=fn...` **yields** $\Theta = \{'e\ \mapsto 'f\ /'a\ \}$ *because* `q` *is a function;*

`([],y)` **yields** $\Theta = \{'g\ list * 'd\ /'e\ \}$ *because* `([],y)` *is the argument of* `q`*;*

`=> y` **yields** $\Theta = \{'d\ /'f\ \}$ *because* `y` *is the result of* `q`*;*

`x::xs` **yields** $\Theta = \{'b\ list/'c\ \}$ **and** $\Theta = \{'g\ /'b\ \}$

because `x::xs` *must be a list and because the types of the two argument patterns for* `q` *must unify;*

`=> (1::(q ...));` **yields** $\Theta = \{int\ list/'d\ \}$

because lists must be homogeneous and because the type must unify with the return type of `q`*.*

Composing all the above unifiers we obtain the parametric type

$q : 'a\ = 'g\ list * int\ list \mapsto int\ list$.

Note how using unification has removed the need to find an ordering in which to analyze the types.

Summary

We have looked at Type handling **in semantic analysis**

- Type inference / reconstruction for ML-like languages
- Hindley-Milner type system

Homework

- **Perform a full, detailed type reconstruction for**

```
fun q [] y = y
  | q (x::xs) y = x::(q xs y);
fun r x = q [1] x;
```