

Programming Languages, F2002

Lecture 5, Wednesday 6 March 2002

- Higher-order functions in Standard ML
- Interpretation of a higher-order functional language (micro-SML)
- Polymorphic type inference in Standard ML
- Types and type schemes
- Type rules
- Unification via the union-find algorithm
- Binding levels for type variable generalization
- Parametric polymorphism in an object-oriented language: Generic C#

IFC

Programming Languages, F2002

Page 5-1

SML: A general fold function for lists

Recursion on the structure of a list can be expressed using `foldr`:

```
fun foldr f e [] = e
  | foldr f e (x::xr) = f(x, foldr f e xr);
```

Therefore many functions can be expressed using `foldr`:

```
fun len xs = foldr (fn (_, res) => 1+res) 0 xs;
fun sum xs = foldr (fn (x, res) => x+res) 0 xs;
fun prod xs = foldr (fn (x, res) => x*res) 1 xs;
fun map g xs = foldr (fn (x, res) => g x :: res) [] xs;
fun concat xss = foldr (fn (xs, res) => xs @ res) [] xss;
```

Other datatypes also have fold functions, e.g. polymorphic trees:

```
datatype 'a tree =
  Lf
  | Br of 'a * 'a tree * 'a tree;
fun tfold f e Lf = e
  | tfold f e (Br(v,t1,t2)) = f(v, tfold f e t1, tfold f e t2);
fun sumtree t = tfold (fn (v, r1, r2) => v + r1 + r2) 0 t;
```

Fold functions resemble visitors (from the 'visitor pattern' of OOP) somewhat.

IFC

Programming Languages, F2002

Page 5-3

SML: Higher-order functions

Applying a function to all elements of a list; anonymous functions (fn):

```
fun map f [] = []
  | map f (x::xr) = f x :: map f xr;
fun doubl x = 2 * x;
map doubl [4, 5, 89];
map (fn x => 2 * x) [4, 5, 89];
```

Repeated application of a function; functions are just values:

```
fun tw g x = g (g x);
val quad = tw doubl;
fun rep n g x = if n=0 then x else rep (n-1) g (g x);
val tw = rep 2;
```

Selecting list elements that satisfy predicate p:

```
fun filter p [] = []
  | filter p (x::xr) = if p x then x :: filter p xr else filter p xr;
val positive = filter (fn i => i mod 2 = 0) [4, 6, 5, 2, 54, 89];
```

IFC

Programming Languages, F2002

Page 5-2

A higher-order functional language

The (concrete and) abstract syntax is as for the one-argument first-order language:

```
datatype expr =
  CstI of int
  | CstB of bool
  | Var of string
  | Let of string * expr * expr
  | Prim of string * expr list
  | If of expr * expr * expr
  | Letfun of string * string * expr * expr (* (f, x, fbody, ebody) *)
  | Call of expr * expr
```

The representation of runtime values is the same too:

```
datatype value =
  Int of int
  | RClo of string * string * expr * vFenv (* (f, x, body, bodyenv) *)
  withtype vFenv = (string, value) env
```

In the higher-order language, a multi-argument function `let f x y = x + y in (f 7) 2 end` can be encoded as `let f x = let fl y = x + y in fl end in (f 7) 2 end`.

IFC

Programming Languages, F2002

Page 5-4

Evaluation of a higher-order functional language, part 1

```
fun eval (e : expr) (env : vFenv) : value =
  case e of
  | CstI i => Int i
  | CstB b => Int (bool2int b)
  | Var x => lookup env x
  | Prim(ope, [e1, e2]) =>
    (case (eval e1 env, eval e2 env) of
     (Int i1, Int i2) =>
       (case ope of
        "*" => Int(i1 * i2)
        "+" => Int(i1 + i2)
        "-" => Int(i1 - i2)
        "=" => Int (bool2int (i1 = i2))
        "<" => Int (bool2int (i1 < i2))
        _ => raise Fail "unknown primitive")
      )
     | _ => raise Fail "eval Prim: unknown arity")
  | Prim _ => raise Fail "eval Prim: unknown arity"
  | Let(x, ehrs, ebody) =>
    let val xval = eval ehrs env
        val env1 = bind1 env (x, xval)
    in eval ebody env1 end
  | ...
```

ITC

SML: Polymorphic type inference in Standard ML

It is hard to use higher-order functions correctly without help from a static (compile-time) types. Higher-order functions are often rather general and deserve polymorphic types.

An example of SML type inference:

```
let fun tw g y = g (g y)
in
  let fun doubl y = 2 * y
  in
    tw doubl 11
  end
end
```

The type found for `tw` is `('a -> 'a) -> ('a -> 'a)`.

The type found for `doubl` is `int -> int`.

The type found for `tw doubl` is `int -> int`.

In fact, the type variable `'a` of `tw` is *generalized* as indicated by the SML compiler:

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

ITC

Evaluation of a higher-order functional language, part 2

```
fun eval (e : expr) (env : vFenv) : value =
  ...
  | If(e1, e2, e3) =>
    (case eval e1 env of
     Int b => if int2bool b then eval e2 env
     else eval e3 env
    )
  | Letfun(f, x, fbody, ebody) =>
    let val env1 = bind1 env (f, RClO(f, x, fbody, env))
    in eval ebody env1 end
  | Call(efun, earg) =>
    (case eval efun env of
     fclOsure as RClO (F, x, fbody, fenv) =>
       let val argv = eval earg env
           val env2 = bind1 fenv (F, fclOsure)
           val env3 = bind1 env2 (x, argv)
       in eval fbody env3 end
     | _ => raise Fail "eval Call: not a function");
```

Only change since last week: the `Call` case no longer requires `efun` to be `Var`.

Now the function `e1` in an application `e1 e2` may be any arbitrary expression.

ITC

SML-style parametric polymorphism

SML-style parametric polymorphism is also called *let-polymorphism*.

It imposes some restrictions:

- Only the types of let-bound variables can be generalized.
- Thus the type of a function parameter is not generalized in the function's body.
- The type of a function is not generalized in the function's own body.

Removing these restrictions seriously complicates type checking.

Type inference becomes undecidable; it cannot always terminate (Henglein; Kfoury, Tiuryn, Urzyczyn: 1990).

ITC

Types and type schemes

A type is `int` or `bool` or a function type $t_1 \rightarrow t_2$ or a type variable α :

$t ::= \text{int} \mid \text{bool} \mid t_1 \rightarrow t_2 \mid \alpha$

Example types: `int` \rightarrow `bool` and $\alpha \rightarrow \alpha$ and $\alpha \rightarrow \beta \rightarrow \alpha$.

Type arrows associate to the right so $\alpha \rightarrow \beta \rightarrow \alpha$ means $\alpha \rightarrow (\beta \rightarrow \alpha)$.

A type scheme has form

$\sigma ::= \forall \alpha_1, \dots, \alpha_n. t$

where the \forall means 'for all', the $\alpha_1, \dots, \alpha_n$ are type variables, and t is a type.

A type scheme $\forall \alpha_1, \dots, \alpha_n. t$ with a non-empty list of type variables represents a polymorphic type.

A type scheme $\forall(). t$ with an empty list of type variables is an ordinary monomorphic type.

Example type schemes: $\forall(). \text{int} \rightarrow \text{bool}$ and $\forall \alpha. \alpha \rightarrow \alpha$ and $\forall \beta. \alpha \rightarrow \beta \rightarrow \alpha$.

TF-C

Generalization

The type of `tw` in

```
fun tw g y = g (g y)
```

is inferred to be $(\text{'a} \rightarrow \text{'a}) \rightarrow (\text{'a} \rightarrow \text{'a})$.

Nothing restricts the type variable `'a` so it is generalized

in the type scheme $\forall \text{'a}. (\text{'a} \rightarrow \text{'a}) \rightarrow (\text{'a} \rightarrow \text{'a})$ for `tw`.

When not to generalize type variables

One cannot always generalize a type variable.

For instance, here the type of `x` is constrained to equal that of `y`, bound further out:

```
let g y = let f x = (x=y) in f 1 & f false end
in g 2 end
```

ILL-TYPED!

Therefore `F` has type scheme $\forall(). \text{'a} \rightarrow \text{bool}$, in which the type variable `'a` has not been generalized.

Generalizing `'a` would be wrong: it would require `y` to have type `int` and `bool` at the same time.

TF-C

Circular types are forbidden

In ML-style type systems, all types must be finite and non-circular.

Therefore, this program is ill-typed:

```
fun g f = f f
```

ILL-TYPED!

It would require `f` to have a 'type' t for which $t = t \rightarrow t_2$ for some t_2 .

Similarly, this is ill-typed:

```
fun g x = g g
```

ILL-TYPED!

It would require `g` to have a 'type' t for which $t = t \rightarrow t_2$ for some t_2 .

TF-C

Modelling types and type schemes in SML

Types and type schemes are modelled in SML using these datatypes:

```
type typevar = ... (* see later *)
datatype typ =
  TYPI
  | TYPB
  | TYPF of typ * typ
  | TYPV of typevar
datatype typescheme =
  TypeScheme of typevar list * typ; (* type parameters and type *)
```

The type scheme $\forall \alpha. \alpha \rightarrow \alpha$ is represented by `TypeScheme([α], TYPF(TYPV(α), TYPV(α))).`

TF-C

Type rules for a higher-order functional language

The type environment $\rho = [x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n]$ maps variable names x to type schemes σ .
 The judgement $\rho \vdash e : t$ asserts that in type environment ρ , the expression e has type t .
 Below, i is an integer constant, b a boolean constant, x a variable, and e_1, e_2 , and so on are expressions.
 The notation $[t_1/\alpha_1, \dots, t_n/\alpha_n]t$ means that α_i is replaced by t_i in t for all i .
 Thus $[\text{int}/\alpha](\alpha \rightarrow \alpha)$ is $\text{int} \rightarrow \text{int}$.

$$\begin{array}{c} \rho \vdash i : \text{int} \\ \rho \vdash b : \text{bool} \\ \frac{\rho(x) = \forall \alpha_1, \dots, \alpha_n. t}{\rho \vdash x : [t_1/\alpha_1, \dots, t_n/\alpha_n]t} \\ \frac{\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int}}{\rho \vdash e_1 + e_2 : \text{int}} \\ \frac{\rho \vdash e_1 : \text{int} \quad \rho \vdash e_2 : \text{int}}{\rho \vdash e_1 < e_2 : \text{bool}} \end{array}$$

IFC

Type inference algorithms in practice

Central ideas in efficient type inference:

- Do not guess the types t_1, \dots, t_n to instantiate with in rule 3. Instead instantiate with new type variables β_1, \dots, β_n .
 Later these new type variables may be equated with other types. This relies on unification, which in turn relies on the union-find algorithm.
- Do not look through the type environment ρ to find free type variables in the Let rules. Instead, remember for each type variable the level (depth of let-bindings) at which it was introduced. When equating two type variables, adjust their binding levels to the lowest (outermost) of the two. If the level of a type variable is lower than the current level, then it is free in the type environment. In that case, do not generalize it.

IFC

$$\frac{\rho \vdash e_1 : t_1 \quad \rho[x \mapsto \forall \alpha_1, \dots, \alpha_n. t_1] \vdash e_2 : t \quad \alpha_1, \dots, \alpha_n \text{ not free in } \rho}{\rho \vdash \text{Let } x = e_1 \text{ in } e_2 \text{ end} : t}$$

$$\frac{\rho \vdash e_1 : \text{bool} \quad \rho \vdash e_2 : t \quad \rho \vdash e_3 : t}{\rho \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : t}$$

$$\frac{\rho[x \mapsto t_x, f \mapsto t_x \rightarrow t_x] \vdash e_r : t_r \quad \rho[f \mapsto \forall \alpha_1, \dots, \alpha_n. t_x \rightarrow t_r] \vdash e_p : t \quad \alpha_1, \dots, \alpha_n \text{ not free in } \rho}{\rho \vdash \text{Let } f x = e_r \text{ in } e_p : t}$$

$$\frac{\rho \vdash e_1 : t_x \rightarrow t_r \quad \rho \vdash e_2 : t_x}{\rho \vdash e_1 e_2 : t_r}$$

The requirement $\alpha_1, \dots, \alpha_n$ not free in ρ means that the type variables must not be constrained by the context. If the type variables are constrained by the context, they cannot be generalized.

IFC

Unification

Equations between types are solved by *unification* (Robinson 1965).

The unification unify $t_1 t_2$ of types t_1 and t_2 is performed as follows, depending on the form of the types:

t_1	t_2	Action
int	int	No action needed
bool	bool	No action needed
$t_{11} \rightarrow t_{12}$	$t_{21} \rightarrow t_{22}$	Unify t_{11} with t_{21} , and unify t_{12} with t_{22}
α	α	No action needed
α	β	Make α equal to β
α	t_2	Make α equal to t_2 , provided α does not occur in t_2
t_1	α	Make α equal to t_1 , provided α does not occur in t_1
All other cases		Unification fails

IFC

Efficient unification uses the *union-find algorithm*

The union-find algorithm maintains a graph, whose nodes are types (including type variables).

Each connected component of the graph is a collection of types that must be equal.

Each connected component has a canonical representative: all other nodes in the component point towards it.

To test whether two nodes are equal (same class): **find** their canonical nodes and see whether they are identical.

When unifying two types, (1) check for circularity, and if OK then (2) union the two equivalence classes.

To form the **union** of two equivalence classes, make one's canonical node point to the other's.

Path compression: after finding the canonical representative of a node, make the node point directly to it.

The amortized time cost of a sequence of union and find operations is almost linear (Tarjan 1984).

IFC

Type variable representation for efficient type inference

In the micro-SML type inference, a type variable is a reference to a pair (roughly, an object with two fields):

- A *link* field, possibly pointing to another node in the type variables in the `Level` rules-union-find graph. If the link is `NO_LINK` then the type variable remains uninstantiated.
- If the link field is `LinkTo t`, then the type variable has been equated with type `t`.
- A *level* field that indicates the binding level of the type variable.

IFC

Some auxiliary functions for the micro-SML type inference

- `specialize lvl ts` returns a type created from the type scheme `ts`.

The type variables $\alpha_1, \dots, \alpha_n$ of `ts` have been replaced with fresh type variables whose levels is `lvl`.

- `generalize lvl t` returns a type scheme created from type `t`.

Only type variables in `t` with higher levels than the current one are being generalized.

IFC

Type inference for micro-SML, part 1

```
fun typ (lvl : int) (venv : (string, typescheme) env) (e : expr) : typ =
  case e of
  | CstI i => TypI
  | CstB b => TypB
  | Var x => (specialize lvl (lookup venv x)
             handle Subscript => raise Fail ("unknown var " ^ x))
  | Prim(ope, [e1, e2]) =>
      let val t1 = typ lvl venv e1
          val t2 = typ lvl venv e2
      in fun chk ta tb tr = (unify ta t1; unify tb t2; tr)
        case ope of
        | "*" => chk TypI TypI TypI
        | "+" => chk TypI TypI TypI
        | "-" => chk TypI TypI TypI
        | "=" => (unify t1 t2; TypB)
        | "<" => chk TypI TypI TypB
        | "&" => chk TypB TypB TypB
        | _ => raise Fail "unknown primitive"
      end
  | Let(x, erns, ebody) =>
      let val lvl = lvl + 1
          val trns = typ lvl venv erns
          val venvbody = bindl venv (x, generalize lvl trns)
      in typ lvl venvbody ebody
      end
  | ...
```

IFC

Type Inference for micro-SML, part 2

```

| ...
| If(e0, e1, e2) =>
    let val t1 = typ lvl venv e1
      val t2 = typ lvl venv e2
    in
      unify TypB (typ lvl venv e0);
      unify t1 t2;
    t1
    end
| Letfun(f, x, fbody, ebody) =>
    let val lvl1 = lvl + 1
      val fty = TypV(newTypeVar lvl1)
      val venv1 = bindl venv (f, TypeScheme([], fty))
      val xty = TypV (newTypeVar lvl1)
      val venvf = bindl venv1 (x, TypeScheme([], xty))
      val rty = typ lvl1 venvf ebody
      val _ = unify fty (TypF(xty, rty))
      val venvbody = bindl venv (f, generalize lvl fty)
    in
      typ lvl venvbody ebody
    end
| Call(e, earg) =>
    let val fty = typ lvl venv e
      val xty = typ lvl venv earg
      val rty = TypV(newTypeVar lvl)
    in
      unify fty (TypF(xty, rty));
    end
end

```

Generics: parametric polymorphism in object-oriented languages

When we use collection classes, Java or C# provide no compile-time type safety:

```

LinkedList names = new LinkedList();
names.add(new Person("Kristen"));
names.add(new Person("Bjarne"));
names.add(new Integer(1998));
names.add(new Person("Anders"));
...
Person p = (Person)names.get(2); // Cast needed, may fail at runtime

```

The compile-time list element type is Object, not Person.

This is because collection classes must work for all types of elements.

Generics can make general code typesafe

```

LinkedList<Person> names = new LinkedList<Person>();
names.Add(new Person("Kristen"));
names.Add(new Person("Bjarne"));
names.Add(new Integer(1998));
names.Add(new Person("Anders"));
...
Person p = names.get(2); // No cast needed

```

The compile-time list element type is Person.

Errors are found at compile-time, not in front of the user. No run-time casts are needed, so the program is faster.

Generic C# example: doubly linked list implementation

```

public class LinkedList<T> : IList<T> {
    Node<T> first, last; // Invariant: first==null iff last==null
    private class Node<T> { // Nested class
        public Node<T> prev, next;
        public T item;
    }
    public T Get(int n) { ... }
    public bool Add(T item) { ... }
    public override bool Equals(object that) { // Inherited from object
        if (that is IList<T>) { // Exact instanceof check
            ...
        }
    }
}

```

Method Get(int) returns a T, not an object.

Efficiency benefits of generics in Generic C#: quicksort

Description	General	Typesafe	Generics	Ints	Strings
Object-based, interface IComparable	yes	no	no	4.99	3.18
Object-based, class OrderedInt	yes	no	no	3.08	2.58
Generic with typed CompareTo	yes	yes	yes	2.45	2.54
Generic with Compare method	yes	yes	yes	1.14	2.19
Hand-specialized, inline <	no	yes	no	0.47	2.10
Hand-specialized with Compare method	no	yes	no	1.06	2.19

Random ints (1,000,000) or strings (200,000); average time/s of 20 runs: 1 GHz P-III; Windows XP; Generic CLR.

- Generics is the only way to have generality, type safety, and efficiency at the same time.
- The only overhead in generics (1.14 vs 0.47) is due to the passing of the Compare method (generally).
- The generics win is clearly larger for the value type int than for the reference type string.

For more examples of using Generic C#, see <http://www.dina.kvl.dk/~sestoft/gcsharp/>