Types

COMP 320

Basic Concepts

An evaluator processing an expression such as

```
if 0 then 3 else 5
```

will get stuck and raise an exception

Typing will ensure we never evaluate these expressions

Typing approximates what happens during runtime

I allows us to detect errors early and to give us precise error messages.

We introduced the basic ideas when we looked at the small language we started with

Type Checking

Recall our grammar:

```
op ::= + | - | * | = | < | orelse
 e ::= n | true | false | e op e
     | if e then e else e | (e)
```

There are only two basic types, int and bool

We denote these as

```
      Types T : int | bool
```

The type int is made up of numbers and the type bool of the values true and false

Types

We use the following notation:

```
e:T means expression e has type T
```

To formally define the type of an expression we use structural induction on the structure of the expression

We use the notation of premises and conclusions that was introduced above in formalizing the operational semantics

The type rules for the primitive values have no premises but only conclusions and we can name the rules for convenience:

```
n : int (T-NUM)
true : bool (T-TRUE)
false : bool (T-FALSE)
```

Type Inference Rules

When is e1+e2 well typed?

```
Premises: e1 : int, e2: int
Conclusion e1+e2 : int
Name: T-PLUS
```

We have similar rules for each of the primitive operations

For if expressions we have:

```
Premises: e:bool, e1:T, e2:T
Conclusion: if e then e1 else e2:T
Name: T-IF
```

Adding Tuples

We can introduce tuples to our language with the productions:

```
e ::= … | (e1, e2) | fst e | snd e
```

We can extend the evaluation rules to allow for tuples as values in expressions

For type checking we introduce product types written T1 x T2

Type checking rules

```
1) Premises (T-PAIR)
     e1:T1,  e2:T2
   Conclusion
     (e1, e2) : T1 x T2

2) Premise (T-FST)
     e : T1 x T2
   Conclusion
     fst e : T1

3) Premise (T-SND)
     e : T1 x T2
   Conclusion
     snd e : T2
```

Let Expressions

```
let x = 5 in x + 3 end
```

We would like to make the following argument:

- 5 is type int, so x will be bound to an integer at runtime
- Assuming x has type int, x + 3 has type int since each subexpression has type int
- The expression has type int

We see that we need a way to reason about the type of a variable

We introduce a "context", denoted Γ, to keep track of assumptions we make about the types of variables

The assumption "variable x has type T" is written x:T

Context

We can define a context inductively

```
        Context Γ ::= . | Γ, x: T
```

The . represents the empty context with no type assumptions.

`Γ |- e:T` means “Given assumption Γ, e has type T”

Type inference rules:

- In all of our previous rules, there is no change except that the context is added as an extra parameter

For example, when is e1+e2 well typed?

```
  Premises: Γ |- e1 : int, Γ |- e2: int
  Conclusion Γ |- e1+e2 : int
  Name: T-PLUS
```

Variables and Let

For variables, we check the context for a type of the variable:

Inference rule:

Premise:

1. `Γ(x) = T`

Conclusion

- `Γ |- x:T`

Name

- T-VAR

Variables and Let

It is the let-expression that introduces bindings. For the let-expression we have the following type inference rule:

Premises:

1. `Γ |- e1:T1`
2. `Γ, x:T1 |- e2:T`

Conclusion

- `Γ |- let x = e1 in e2 end : T`
- `x must be a new varaiable`

`Name T-LET`

The variable x must be new to ensure that the declaration x:T1 does not clash with any other declaration

We can rename the bound variable x in e2 if there is a clash

Examples

We can show that

```
. |- let x=5 in x+3 end : int
```

A more complicated example

```
. |- let x=5 in (let y=x+2 in x+y end) end :int
```

Type Inference

Type checking: given expression, e and type T, verify that e:T

However in let expressions, we don't know what the type of e1 is. We have to infer it

Type inference: Given e, infer a type T such that e:T

Can this be done. In our language so far, yes

Uniqueness: if `Γ |- e:T` and `Γ |- e:T',` then `T = T'`

Functions

We add a new descriptor for types:

```
    T ::= int | bool | T x T | T -> T
```

A function fn x => e has type T1 -> T2 if, assuming that x has type T1, we can show that e has type T2

We can give formal inference rules for function definition and for function application

Rules for Function Definition

Function definition:

Premise:

1. `Γ, x:T1 |- e:T`

Conclusion:

- `Γ |- fn x => e : T1 -> T2`
- `x must be a new varaiable`

`Name: T-FN`

Rules for Function Application

Function applicatiom:

Premise:

1. `Γ |- e1:T1 -> T`
2. `Γ |- e2:T1`

Conclusion:

- `Γ |- e1 e2 : T`

`Name: T-APP`

Uniqueness?

What is the type of

```
        fn x => x
```

It has type `int -> int`

But it also has types:

- `bool -> bool`
- `(bool -> int) -> (bool -> int)`
- `etc`

Principal Type

What is the type of

```
        fn x => x
```

We can add type variables to allow for expressions having multiple types:

```
        T ::= int | bool | T x T | T -> T | 'a
```

The function has all of the types mentioned above

The **principal type** is the most general type that covers all of the previous examples and can be expressed as `'a -> 'a`

Informal

We begin by giving an informal description of type inference using examples from SML

In ML almost every expression has a principal type scheme

That is there is a most general way to infer types that maximizes generality

ML inference

We have seen that `(fn x => x)` has principal type `‘a->’a`

We have to consider the context in which it is used as well

`(fn x => x) (5)` forces the type of the identity function to be `int->int`

`(fn x => x) (fn x => x) (5)` forces the first occurrence of the identity function to be `(int->int) ->( int->int)`

The second occurrence of the identity function is of type `int->int`

This inference is achieved by a process of constraint satisfaction

Informal Description

Type Inference methodology:

1. Assign a new type variable to each subexpression
2. Each use of an expression may imply some property of its type: this is a type constraint. We build a set of type constraints from the smallest expressions to progressively larger ones.
3. Solve the constraints

Informal Description

Solving the contraints has three possible results

- Overconstrained: the type constraints cannot be satisfied
- Underconstrained: the type constraints are satisfiable, but there is more than one solution:
  - Polymorphic: The type is well-formed type and can be written using type variables.
  - Ambiguous: There is more than one way to satisfy the constraints. In this case, SML will sometimes arbitrarily choose one of the solutions --- for example, fn (x, y) => x + y is assumed to have type (int * int) -> int.
- Uniquely determined

This methodology of constraint satisfaction makes it difficult to pinpoint the error in a program that is not well types

It is in general impossible to attribute the breakdown in the algorithm for finding constraints to a particular constraint, making the reporting of type checking errors very obscure

Example 1 – Uniquely Determined

```
fun foo (w, x, y, z) =
     if y then w::tl(x) else y::z;
```

Assign a type variable: foo : 'a

Since foo is used as a function we have a type constraint: 'a = 'b -> 'c

The argument is a 4-tuple so we have constraints: we assign type variables to the argument names:

```
'b = ('d * 'e * 'f * 'g)
w : 'd
x : 'e
y : 'f
z : 'g
```

Example 1 – Uniquely Determined

```
fun foo (w, x, y, z) =
     if y then w::tl(x) else y::z;
```

y is used as the condition in a if statement, so `y : 'f = bool`

We apply the function tl to x so x must be a list:

```
    x : 'e = 'h list
    tl(x) : 'h list
```

Since we cons w onto tl(x), `w : 'd = 'h`

From the expression y::z, we have `z : 'g = 'i list`

Now `y : ‛f = ‛I` and `y : bool`, so `z : bool list`

Example 1 – Uniquely Determined

```
fun foo (w, x, y, z) =
     if y then w::tl(x) else y::z;
```

The branches of an if statement must have the same type. This allows us to propagate and solve further type equalities:

```
w::tl(x) : 'h list = bool list
tl(x) : 'h list = bool list
x : 'e = 'h list = bool list
w : 'd = bool
```

Example 1 – Uniquely Determined

```
fun foo (w, x, y, z) =
    if y then w::tl(x) else y::z;
```

We can now deduce the type of the entire if expression, yielding:

```
      (if y then w::tl(x) else y::z) : bool list
```

The complete type of the function is then:

```
   myFun : bool * bool list * bool * bool list -> bool list
```

Inferring a Polymorphic Type

In the next example, we infer a polymorphic type

We do it in a little less detail than the previous example

```
fun foo w1 nil _ = (w1, nil)
  | foo w2 (v::x) y =
      let
        val bar = (w2, y v);
        val bif = v :: tl(x);
      in
        (#1(bar), bif)
      end;
```

Example 2

```
fun foo w1 nil   _ = (w1, nil)
  | foo w2 (v::x) y = . . .
```

foo is a curried function so we can assign type variables to its three curried arguments and to its return value:

```
 foo : 'a -> 'b -> 'c -> 'd
```

Example 2

```
fun foo w1 nil _ = (w1, nil) . . .
```

From the first case of this function's definition, we can derive the following (note we assign a fresh type variable to the second argument's element type):

```
w1 : 'a
'b = 'e list
```

The first case of the function returns a tuple, whose first element is w and whose second element is some kind of list; therefore:

```
'd = 'a * 'f list
```

Example 2

```
fun foo w1 nil _ = (w1, nil)
  | foo w2 (v::x) y = . . .
```

Now we examine the second case of the function. From its parameters:

```
    w2 : 'a
    (v::x) : 'b = 'g list
    v : 'g
    x : 'g list
    y : `c
```

However, we already know from the first case that 'b = 'e list, so we further derive

```
    'g list = 'b = 'e list
```

Example 2

```
let
 val bar = (w2, y v);
        val bif = v :: tl(x);
      in
        (#1(bar), bif)
      end;
```

Examining the bar binding, we observe that y is used as a function, and applied to v, so

```
     y : `c = 'g -> 'i
     bar : 'a * ('g -> 'i)
```

Examining the bif binding, we obtain:

```
     v : 'g
     tl(x) : 'g list
     bif : 'j = 'k list = 'g list
```

Example 2

```
let
 val bar = (w2, y v);
        val bif = v :: tl(x);
      in
        (#1(bar), bif)
      end;
```

Finally, examining the body of the let-expression, we obtain the type of the result: `'a * 'g list`.

Since this expression determines the return type of foo, we get

```
'd = 'a * 'g list = 'a * 'f list
```

Example 2

If we solve all the accumulated constraints, we get

```
    foo : 'a -> 'g list -> ('g -> 'i) -> 'a * 'g list
```

To improve readability, the SML/ML interpreter will consistently rename the type variables in polymorphic types so that they occur in consecutive alphabetical order, yielding:

```
    foo : 'a -> 'b list -> ('b -> 'c) -> 'a -> 'b list
```

Unification

Consider the expression:

```
fun ... = ...
        val (x, y) = (3, z)
```

where z is defined earlier in the function. Suppose that we had assigned the fresh type variable 'a to z. Our first step in inferring the types of x and y would be to set up a type equation like the following:

```
('b * 'c) = (int * 'a)
```

We must attempt to unify the left- and right-hand sides of this equation.

Although the result seems obvious in this case, we will consider a general procedure for preforming this ‘unification’

Unification (simplified)

A function in ML-like pseudocode that determines whether two values can be unified:

```
fun unify(t1, t2) =
   if t1 and t2 are base types then
      if t1 = t2 then true else false
   else if t1 is a type variable
           or t2 is a type variable then
      return true (adding constraint t1 = t2)
   else if t1 and t2 are compound types then
      if t1 and t2 have different constructors then
          false
      else
          if each corresponding element of t1 and t2 can be unified then
             true
          else
             false
```

Unification

This is a simplification that only returns true or false.

Real unification returns a set of type equalities resulting from successful unifications.

Nevertheless, it gives us the intuition for how to unify the tuple example above:

In the example, the constructors have the same kind (tuple) and arity (2).

Recursively attempt to unify the corresponding positions in each tuple:

- unify('b, int): returns true, with type constraint 'b = int.
- unify('c, 'a): returns true, with type constraint 'c = 'a.

Since all recursive-unifications succeed, return true (and keep all type equalities accumulated while performing recursive unification).

The unified type is int * 'a.

Another Example

```
'a * ('b * 'c) * 'c list = ('v * 'w) * ('x * 'y) * 'z
```

The top-level type constructors are 3-tuples, and produce the following unifications:

```
unify('a, (‘v * ‘w))
unify(('b * 'c), ('x * 'y))
unify('c list, 'z)
```

For the first and third of these, at least one of the conclusions is a variable, so we can’t go further, but for the second we can proceed and recursively unify:

```
unify('b, 'x)
unify('c, 'y)
```

Example (cont’d)

We end up with the following type equations:

```
'a = ‘v * ‘w
'b = 'x
'c = 'y
'c list = 'z
```

Solving all these equations, the unified type is:

```
        ('v * 'w) * ('b * 'c) * 'c list
```

SML/NJ, the would rename this to

```
        ('a * 'c) * ('d * 'e) * 'e list
```

General Bounded Polymorphism

General type variables are universally quantified

That is, they vary over every possible type

This can cause difficulties for type inference because of the fact that some operators may be overloaded but not of general type

For example + could be `int*int -> int` or `real*real -> real`.

It is overloaded but not of type `‘a * ‘a -> ‘a`

Options for type checking:

1. The expression is ambiguous and is rejected until the programmer provides explicit type information
2. Choose a default interpretation (say int) and issue a warning

Overloading

Sometimes the type can be inferred from the context:

```
(fn x => x+x) 3
(fn x => x+x) 3.0
```

But it is impossible to infer a type for the function in

```
let
  val double = fn x => x+x
In
  (double 3, double 3.0)
```

Equality types

The SML type checker choses simplicity over a more complex analysis that would take this into account. It does not fully support such bounded polymorphism.

An exception in which ML does support one form of bounded polymorphism is found in equality types.

An equality type is either:

1. Some base type (e.g., int or string) that supports the equality operator =.
2. Some type constructed only by applying constructors to other types that are (recursively) equality types --- e.g., int * string, bool list, or (int * bool) list.

An equality type variable is written with two quotes, e.g. ''a

We have seen examples of such types earlier (and in an asssignment)

More Formally

We can define the language of types (as before)

```
T ::= int | bool | T x T | T -> T | 'a
```

When we apply the identity function (of type `'a -> 'a)` to an integer we instantiate the type of the function by replacing ‘a with int.

This is essentially a substitution which we can define similar to what we did earlier.

Substitution Rules

$[T/\alpha](\alpha) = T$

$[T/\alpha](\beta) = \beta$

$[T/\alpha](\text{int}) = \text{int}$

$[T/\alpha](\text{bool}) = \text{bool}$

$[T/\alpha](T1 \times T2) = [T/\alpha]T1 \times [T/\alpha]T2$

$[T/\alpha](T1 \to T2) = [T/\alpha]T1 \to [T/\alpha]T2$

We let σ represent a set of simultaneous type substitutions

Theorem: If `Γ |- e:T` then $[\sigma]$ `Γ |- e:` $[\sigma]$ `T`

Type Inference

We ask whether there is some instantiation of a type variable such that the expression has that type

As we saw in the examples, we generate constraints of the form T1=T2

For example in the expression `if e then e1 else e2`

1. Infer type T for e
2. Infer type T1 for e1
3. Infer type T2 for e2

The constraints will be:

- T = bool
- T1 = T2

Generating Constraints

We use the notation `Γ |- e2 => T/C` to mean we can infer type T for e in the given type environment, modulo satisfying a set of constraints C

We generate constraints using inference rules

There are no constraints for number, Boolean and variables, so we can write rules such as

For example for numbers (and similarly for Booleans true and false):

Premise:
- None

Conclusion
- `Γ |- n => int/Ø`

Name
- B-NUM

Generating Constraints

For variables, we check the context for a type of the variable:

Inference rule:

Premise:

1. `x:T ∈Γ`

Conclusion

- `Γ |- x => T/Ø`

Name

- B-VAR

If Expressions

Premises:

1. `Γ |- e => T/C`
2. `Γ |- e1 => T1/C1`
3. `Γ |- e2 => T2/C2`

Conclusion:

- `Γ |- if e then e1 else e2 => T1/C ∪ C1 ∪ C2 ∪ {T=bool, T1=T2}`

`Name: B-IF`

The rule for let-expression is easy

Primitive Operators

We give rules for Plus, rules for others are similar

Premises:

1. `Γ |- e1 => T1/C1`
2. `Γ |- e2 => T2/C2`

Conclusion:

- `Γ |- e1 + e2 => int/C1 ∪ C2 ∪{T1=int, T2=int}`

`Name: B-PLUS`

Function Definition

For functions we don’t know the type of the input argument before the function is applies. We generate a new type variable ‘a

Premise:

1. `Γ, x:’a |- e => T/C`

Conclusion:

- `Γ |- fn x -> e => ’a -> T/C`

`Name: B-FN`

Function Application

Premises:

1. `Γ |- e1 => T1/C1`
2. `Γ |- e2 => T2/C2`

Conclusion:

- `Γ |- e1 e2 => 'a/C1 ∪ C2 ∪{T1=T2 -> 'a}`

`Name: B-APP`

Solving Type Constraints

T1 and T2 are unifiable if there is an instantiation σ for the free type variables in T1 and T2 such that [σ]T1 = [σ]T2

Unification is an algorithm to determine whether two objects can be made syntactically equal

A more formal description of the simplified unification that only tests whether a set of constraints is unifiable

Unification (simplified)

```
{C, int = int} => C

{C, bool = bool} => C

{C, (T1->T2) = (S1->S2)} => {C, T1=S1, T2=S2}

{C, α=T} => [T/α]C where α is not in FV(T)
```

Using these rules we can transform a given set of contraints

The algorithm terminates when no rules are applicable

It succeeds if we have derived the empty set. Otherwise it fails.