Types, Type Inference and Unification

Mooly Sagiv

Slides by Kathleen Fisher and John Mitchell

Cornell CS 6110
Summary (Functional Programming)

• Lambda Calculus
• Basic ML
• Advanced ML: Modules, References, Side-effects
• Closures and Scopes
• Type Inference and Type Checking
Outline

• General discussion of types
  – What is a type?
  – Compile-time versus run-time checking
  – Conservative program analysis

• Type inference
  – Discuss algorithm and examples
  – Illustrative example of static analysis algorithm

• Polymorphism
  – Uniform versus non-uniform implementations
Language Goals and Trade-offs

• Thoughts to keep in mind
  – What features are convenient for programmer?
  – What other features do they prevent?
  – What are design tradeoffs?
    • Easy to write but harder to read?
    • Easy to write but poorer error messages?
  – What are the implementation costs?
What is a type?

• A type is a collection of computable values that share some structural property.

Examples

- int
- string
- int → bool
- (int → int) → bool
- [a] → a
- [a] * a → [a]

Non-examples

- {3, True, \x->x}
- Even integers
  - f:int → int | x>3 => f(x) > x *(x+1)

Distinction between sets of values that are types and sets that are not types is language dependent
Advantages of Types

• Program organization and documentation
  – Separate types for separate concepts
    • Represent concepts from problem domain
  – Document intended use of declared identifiers
    • Types can be checked, unlike program comments

• Identify and prevent errors
  – Compile-time or run-time checking can prevent meaningless computations such as $3 + \text{true} - \text{“Bill”}

• Support optimization
  – Example: short integers require fewer bits
  – Access components of structures by known offset
What is a type error?

• Whatever the compiler/interpreter says it is?
• Something to do with bad bit sequences?
  – Floating point representation has specific form
  – An integer may not be a valid float
• Something about programmer intent and use?
  – A type error occurs when a value is used in a way that is inconsistent with its definition
    • Example: declare as character, use as integer
Type errors are language dependent

• Array out of bounds access
  – C/C++: run-time errors
  – OCaml/Java: dynamic type errors

• Null pointer dereference
  – C/C++: run-time errors
  – OCaml: pointers are hidden inside datatypes
    • Null pointer dereferences would be incorrect use of these datatypes, therefore static type errors
Compile-time vs Run-time Checking

• JavaScript and Lisp use run-time type checking
  – f(x) Make sure f is a function before calling f

• OCaml and Java use compile-time type checking
  – f(x) Must have f: A → B and x : A

• Basic tradeoff
  – Both kinds of checking prevent type errors
  – Run-time checking slows down execution
  – Compile-time checking restricts program flexibility
    • JavaScript array: elements can have different types
    • OCaml list: all elements must have same type
  – Which gives better programmer diagnostics?
Expressiveness

• In JavaScript, we can write a function like

```javascript
function f(x) { return x < 10 ? x : x(); }
```

Some uses will produce type error, some will not

• Static typing always conservative

```javascript
if (complicated-boolean-expression)
  then f(5);
else f(15);
```
Type Safety

• Type safe programming languages protect its own abstractions
• Type safe programs cannot go wrong
• No run-time errors
• But exceptions are fine
• The small step semantics cannot get stuck
• Type safety is proven at language design time
Relative Type-Safety of Languages

• **Not safe: BCPL family, including C and C++**
  – Casts, unions, pointer arithmetic
• **Almost safe: Algol family, Pascal, Ada**
  – Dangling pointers
  • Allocate a pointer p to an integer, deallocate the memory referenced by p, then later use the value pointed to by p
  • Hard to make languages with explicit deallocation of memory fully type-safe
• **Safe: Lisp, Smalltalk, ML, Haskell, Java, JavaScript**
  – Dynamically typed: Lisp, Smalltalk, JavaScript
  – Statically typed: OCaml, Haskell, Java

If code accesses data, it is handled with the type associated with the creation and previous manipulation of that data
Type Checking vs Type Inference

• Standard type checking:

```c
int f(int x) { return x+1; }
int g(int y) { return f(y+1)*2; }
```

  - Examine body of each function
  - Use declared types to check agreement

• Type inference:

```c
int f(int x) { return x+1; }
int g(int y) { return f(y+1)*2; }
```

  - Examine code without type information
  - Infer the most general types that could have been declared

ML and Haskell are *designed* to make type inference feasible
The Type Inference Problem

• Input: A program without types (e.g., Lambda calculus)

• Output: A program with type for every expression (e.g., typed Lambda calculus)
  – Every expression is annotated with its most general type
Why study type inference?

• Types and type checking
  – Improved steadily since Algol 60
    • Eliminated sources of unsoundness
    • Become substantially more expressive
  – Important for modularity, reliability and compilation

• Type inference
  – Reduces syntactic overhead of expressive types
  – Guaranteed to produce most general type
  – Widely regarded as important language innovation
  – Illustrative example of a flow-insensitive static analysis algorithm
History

• Original type inference algorithm
  – Invented by Haskell Curry and Robert Feys for the simply typed lambda calculus in 1958

• In 1969, Hindley
  – extended the algorithm to a richer language and proved it always produced the most general type

• In 1978, Milner
  – independently developed equivalent algorithm, called algorithm W, during his work designing ML

• In 1982, Damas proved the algorithm was complete.
  – Currently used in many languages: ML, Ada, Haskell, C# 3.0, F#, Visual Basic .Net 9.0. Have been plans for Fortress, Perl 6, C++0x,...
Type Inference: Basic Idea

• Example

fun x -> 2 + x
→: int -> int = <fun>

• What is the type of the expression?

  • + has type: int → int → int
  • 2 has type: int
  • Since we are applying + to x we need x : int
  • Therefore fun x -> 2 + x has type int → int
Imperative Example

\[
\begin{align*}
x &:= b[z] \\
an [b[y]] &:= x
\end{align*}
\]
Type Inference: Basic Idea

• Example

```plaintext
fun f => f 3
(int -> a) -> a = <fun>
```

• What is the type of the expression?
  – 3 has type: int
  – Since we are applying f to 3 we need f : int → a and the result is of type a
  – Therefore `fun f => f 3` has type (int → a) → a
Type Inference: Basic Idea

• Example

```plaintext
fun f => f (f 3) (int -> int) -> int = <fun>
```

• What is the type of the expression?
Type Inference: Basic Idea

• Example

```plaintext
fun f => f (f "hi")
(string -> string) -> string = <fun>
```

• What is the type of the expression?
Type Inference: Basic Idea

• Example

fun f => f (f 3, f 4)

• What is the type of the expression?
**Type Inference: Complex Example**

let square = \(z \cdot z \times z\) 
  in 
\(\lambda f. \lambda x. \lambda y.\) 
  if (f x y) 
  then (f (square x) y) 
  else (f x (f x y))

\[ * : \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \]

\[ z : \text{int} \]

\[ \text{square} : \text{int} \rightarrow \text{int} \]

\[ f : a \rightarrow (b \rightarrow \text{bool}), x : a, y : b \]

\[ a : \text{int} \]

\[ b : \text{bool} \]

\[ (\text{int} \rightarrow \text{bool} \rightarrow \text{bool}) \rightarrow \text{int} \rightarrow \text{bool} \rightarrow \text{bool} \]
Unification

• Unifies two terms
• Used for pattern matching and type inference
• Simple examples
  – int * x and y * (bool * bool) are unifiable for y = int and x = (bool * bool)
  – int * int and int * bool are not unifiable
Substitution

Types:
<type> ::= int | float | bool |...
   | <type> → <type>
   | <type> * <type>
   | variable

Terms:
<term> ::= constant
   | variable
   | f(<term>, ..., <term>)

• The essential task of unification is to find a substitution that makes the two terms equal

  \[ f(x, h(x, y)) \{ x \mapsto g(y), y \mapsto z \} = f(g(y), h(g(y), z) \]

• The terms \( t_1 \) and \( t_2 \) are unifiable if there exists a substitution \( S \) such that \( t_1 S = t_2 S \)

• Example: \( t_1 = f(x, g(y)) \), \( t_2 = f(g(z), w) \)
Most General Unifiers (mgu)

• It is possible that no unifier for given two terms exist
  – For example x and f(x) cannot be unified

• There may be several unifiers
  – Example: $t_1 = f(x, g(y))$, $t_2 = f(g(z), w)$
    • $S = \{x \mapsto g(z), y \mapsto w, w \mapsto g(w)\}$
    • $S' = \{x \mapsto g(f(a, b)), y \mapsto f(b, a), z \mapsto f(a, b), w \mapsto g(f(b, a))\}$

• When a unifier exists, there is always a most general unifier (mgu) that is unique up to renaming

• $S$ is the most general unifier of $t_1$ and $t_2$ if
  – It is a unifier of $t_1$ and $t_2$
  – For every other unifier $S'$ of $t_1$ and $t_2$ there exists a refinement of $S$ to give $S'$

• mgu can be efficiently computed
  – $\text{mgu}(f(x, g(y)), f(g(z), w)) = \{x \mapsto g(z), y \mapsto w, w \mapsto g(w)\}$
  – $\text{mgu}(\{y \mapsto g(w)\}, f(x, g(y)), f(g(z), w)) = \{y \mapsto g(w), x \mapsto g(z), w \mapsto g(g(w))\}$
Type Inference with mgu

• Example

```plaintext
fun f => f (f "hi")
(string -> string) -> string = <fun>
```

• What is the type of the expression?

\[
\lambda f : T_1.\text{apply}(f : T_1, \text{apply}(f : T_1, \"hi\" : \text{string}) : T_2) : T_3
\]

\[
\text{mgu}(T_1, \text{string} \to T_2) = \{T_1 \mapsto \text{string} \to T_2\} = S
\]

\[
\text{mgu}(S, T_1, T_2 \to T_3) = \\
\{T_1 \mapsto \text{string} \to T_2, T_2 \mapsto \text{sring}, T_3 \mapsto \text{sring}\}
\]
Type Inference Algorithm

- Parse program to build parse tree
- Assign type variables to nodes in tree
- Generate constraints:
  - From environment: literals (2), built-in operators (+), known functions (tail)
  - From form of parse tree: e.g., application and abstraction nodes
- Solve constraints using unification
- Determine types of top-level declarations
Step 1: Parse Program

• Parse program text to construct parse tree

```
let f x = 2 + x
```

Infix operators are converted to Curried function application during parsing: (not necessary)

```
2 + x  ➜  (+) 2 x
```
Step 2: Assign type variables to nodes

Variables are given same type as binding occurrence

\[ f \ x = 2 + x \]
Step 3: Add Constraints

\[ t_0 = t_1 \rightarrow t_6 \]
\[ t_4 = t_1 \rightarrow t_6 \]
\[ t_2 = t_3 \rightarrow t_4 \]
\[ t_2 = \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \]
\[ t_3 = \text{int} \]

\[ \text{let } f \ x = 2 + x \]
Step 4: Solve Constraints

\[
\begin{align*}
t_0 &= t_1 \rightarrow t_6 \\
t_4 &= t_1 \rightarrow t_6 \\
t_2 &= t_3 \rightarrow t_4 \\
t_2 &= \text{int} \rightarrow (\text{int} \rightarrow \text{int}) \\
t_3 &= \text{int} \\
\end{align*}
\]
Step 5:
Determine type of declaration

\[
\begin{align*}
  t_0 &= \text{int} \rightarrow \text{int} \\
  t_1 &= \text{int} \\
  t_6 &= \text{int} \rightarrow \text{int} \\
  t_4 &= \text{int} \rightarrow \text{int} \\
  t_2 &= \text{int} \rightarrow \text{int} \rightarrow \text{int} \\
  t_3 &= \text{int} \\
\end{align*}
\]

\[
\text{let } f\ x = 2 + x \\
\text{val } f : \text{int} \rightarrow \text{int} = \langle \text{fun} \rangle
\]
Function application (apply f to x)

- Type of f (t_0 in figure) must be domain → range
- Domain of f must be type of argument x (t_1 in fig)
- Range of f must be result of application (t_2 in fig)
- Constraint: t_0 = t_1 → t_2
Constraints from Abstractions

- Function declaration:
  - Type of f (t_0 in figure) must domain $\rightarrow$ range
  - Domain is type of abstracted variable x (t_1 in fig)
  - Range is type of function body e (t_2 in fig)
  - Constraint: $t_0 = t_1 \rightarrow t_2$
Inferring Polymorphic Types

Example:

```
let f g = g 2
val f : (int -> t_4) -> t_4 = <fun>
```

Step 1:
Build Parse Tree
Inferring Polymorphic Types

• Example:

let f g = g 2
val f : (int -> t_4) -> t_4 = fun

• Step 2:
Assign type variables
Inferring Polymorphic Types

- Example:
  ```ocaml
  let f g = g 2
  val f : (int -> t_4) -> t_4 = <fun>
  ```
- Step 3:
  Generate constraints

```
t_0 = t_1 -> t_4
f :: t_0

f = t_0 = t_1

(\theta) :: t_4

g :: t_1

2 :: t_3
```
Inferring Polymorphic Types

- Example:
  - Step 4: Solve constraints

```ocaml
let f g = g 2
val f : (int -> t_4) -> t_4 = <fun>
```

Diagram:

```
t_0 = t_1 -> t_4
f :: t_0

t_1 = t_3 -> t_4
g :: t_1

t_3 = int
(\emptyset) :: t_4

t_0 = (int -> t_4) -> t_4

:: ≡:
```
Inferring Polymorphic Types

• Example:

```ocaml
let f g = g 2
val f : (int -> t_4) -> t_4 = <fun>
```

• Step 5:

Determine type of top-level declaration

Unconstrained type variables become polymorphic types

```
t_0 = (int -> t_4) -> t_4
t_1 = int -> t_4
t_3 = int
```

```
:: ≡ ::
```
Using Polymorphic Functions

• Function:
  ```plaintext
  let f g = g 2
  val f : (int -> t_4) -> t_4 = <fun>
  ```

• Possible applications:
  ```plaintext
  let add x = 2 + x
  val add : int -> int = <fun>
  f add
  :- int = 4
  ```
  ```plaintext
  let isEven x = mod (x, 2) == 0
  val isEven: int -> bool = <fun>
  f isEven
  :- bool= true
  ```
Recognizing Type Errors

- **Function:**
  ```
  let f g = g 2
  val f : (int -> t_4) -> t_4 = <fun>
  ```

- **Incorrect use**
  ```
  let not x = if x then true else false
  val not : bool -> bool = <fun>
  f not
  > Error: operator and operand don’t agree
  operator domain: int -> a
  operand: bool -> bool
  ```

- **Type error:**
  cannot unify bool → bool and int → t
Another Example

- Example:
  ```ml
  let f (g,x) = g (g x)
  val f : ((t_8 -> t_8) * t_8) -> t_8
  ```

- Step 1:
  Build Parse Tree
Another Example

• Example:

\[
\begin{align*}
\text{let} & \ f \ (g,x) = g \ (g \ x) \\
\text{val} & \ f : \ ((t_8 \to t_8) \times t_8) \to t_8
\end{align*}
\]

• Step 2:
Assign type variables
Another Example

Example:

Step 3:
Generate constraints

\[
\begin{align*}
\text{let } f \ (g,x) &= g \ (g \ x) \\
\text{val } f &: ((t\_8 \to t\_8) \times t\_8) \to t\_8
\end{align*}
\]

\[
\begin{align*}
t\_0 &= t\_3 \to t\_8 \\
t\_3 &= (t\_1, t\_2) \\
t\_1 &= t\_7 \to t\_8 \\
t\_1 &= t\_2 \to t\_7
\end{align*}
\]
Another Example

• Example:
  
  • Step 4:
    Solve constraints

  \[
  \begin{align*}
  t_0 &= (t_8 \rightarrow t_8, t_8) \rightarrow t_8 \\
  t_8 &= (t_1, t_2) \\
  t_1 &= t_7 \rightarrow t_8 \\
  \end{align*}
  \]

  \[
  \text{let } f \ (g,x) = g \ (g \ x) \\
  \text{val } f : ((t_8 \rightarrow t_8) \times t_8) \rightarrow t_8
  \]
Another Example

• Example:

• Step 5:
  Determine type of \( f \)

let \( f \) (g,x) = g (g x)
val f : ((t_8 -> t_8) * t_8) -> t_8

t_0 = t_3 -> t_8
t_3 = (t_1 * t_2)
t_1 = t_7 -> t_8
t_1 = t_2 -> t_7

t_0 = (t_8 -> t_8 * t_8) -> t_8
Pattern Matching

• Matching with multiple cases

  ```
  let isempty l = match l with
      | [] -> true
      | _  -> false
  ```

• Infer type of each case
  
  – First case:
  
    ```
    [t_1] -> bool
    ```
  
  – Second case:
  
    ```
    t_2 -> bool
    ```

• Combine by unification of the types of the cases

  ```
  val isempty : [t_1] -> bool = <fun>
  ```
Bad Pattern Matching

• Matching with multiple cases

```ocaml
let isempty l = match l with
  | [] -> true
  | _ -> 0
```

• Infer type of each case

  – First case:

    
    

  – Second case:

    
    

• Combine by unification of the types of the cases

  
  

Type Error: cannot unify bool and int
Recursion

```ocaml
let rec concat a b = match a with
| []  -> b
| x::xs -> x :: (concat xs b)
```

- To handle recursion, introduce type variables for the function:
  ```ocaml
  concat : t_1 -> t_2 -> t_3
  ```

- Use these types to conclude the type of the body:
  - Pattern matching first case:
    ```ocaml
    [t_4] -> t_5 -> t_5
    unify [t_4] with t_1, t_5 with t_2,
    t_5 with t_3
    t_1 = [t_4] and t_2 = t_3 = t_5
    ```
  - Pattern matching second case:
    ```ocaml
    [t_6] -> t_7 -> [t_6]
    unify [t_6] with t_1, t_7 with t_2,
    [t_6] with t_3
    ```
    ```ocaml
    unify [t_6] with t_1, t_7 with t_2,
    t_3 with [t_6]
    ```
Recursion

To handle recursion, introduce type variables for the function:

```ml
let rec concat a b = match a with
| []  -> b
| x::xs -> x :: concat xs b
```

Conclude the type of the function:

```ml
concat : t_1 -> t_2 -> t_3
```

Conclude the type of the function:

```ml
```
Most General Type

• Type inference produces the most general type

```ml
let rec map f arg = function
    [] -> []
  | hd :: tl -> f hd :: (map f tl)
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

• Functions may have many less general types

```ml
val map : (t_1 -> int, [t_1]) -> [int]
val map : (bool -> t_2, [bool]) -> [t_2]
val map : (char -> int, [cChar]) -> [int]
```

• Less general types are all instances of most general type, also called the principal type
Information from Type Inference

• Consider this function...

```plaintext
let reverse ls = match ls with
    [] -> []
| x :: xs -> reverse xs
```

... and its most general type:

```plaintext
:- reverse :: list 't_1 -> list 't_2
```

• What does this type mean?

Reversing a list should not change its type, so there must be an error in the definition of reverse!
Complexity of Type Inference Algorithm

• When Hindley/Milner type inference algorithm was developed, its complexity was unknown

• In 1989, Kanellakis, Mairson, and Mitchell proved that the problem was exponential-time complete

• Usually linear in practice though...
  – Running time is exponential in the depth of polymorphic declarations
Type Inference: Key Points

• Type inference computes the types of expressions
  – Does not require type declarations for variables
  – Finds the most general type by solving constraints
  – Leads to polymorphism

• Sometimes better error detection than type checking
  – Type may indicate a programming error even if no type error

• Some costs
  – More difficult to identify program line that causes error
  – Natural implementation requires uniform representation sizes
  – Complications regarding assignment took years to work out

• Idea can be applied to other program properties
  – Discover properties of program using same kind of analysis
Parametric Polymorphism: OCaml vs C++

• OCaml polymorphic function
  – Declarations (generally) require no type information
  – Type inference uses type variables to type expressions
  – Type inference substitutes for type variables as needed to instantiate polymorphic code

• C++ function template
  – Programmer must declare the argument and result types of functions
  – Programmers must use explicit type parameters to express polymorphism
  – Function application: type checker does instantiation
Example: Swap Two Values

- OCaml

```ocaml
let swap (x, y) =
  let temp = !x in
  (x := !y; y := temp)
val swap : 'a ref * 'a ref -> unit = <fun>
```

- C++

```cpp
template <typename T>
void swap(T& x, T& y){
    T tmp = x;  x=y;  y=tmp;
}
```

Declarations both swap two values polymorphically, but they are compiled very differently.
Implementation

• OCaml
  – `swap` is compiled into one function
  – Typechecker determines how function can be used

• C++
  – `swap` is compiled differently for each instance
    (details beyond scope of this course …)

• Why the difference?
  – OCaml ref cell is passed by pointer. The local `x` is a pointer to value on heap, so its size is constant
  – C++ arguments passed by reference (pointer), but local `x` is on the stack, so its size depends on the type
Polymorphism vs Overloading

• Parametric polymorphism
  – Single algorithm may be given many types
  – Type variable may be replaced by any type
  – if $f : t \rightarrow t$ then $f : \text{int} \rightarrow \text{int}, f : \text{bool} \rightarrow \text{bool}, ...$

• Overloading
  – A single symbol may refer to more than one algorithm
  – Each algorithm may have different type
  – Choice of algorithm determined by type context
  – Types of symbol may be arbitrarily different
  – In ML, $+$ has types $\text{int} \times \text{int} \rightarrow \text{int}, \text{real} \times \text{real} \rightarrow \text{real},$ no others
  – Haskel permits more general overloading and requires user assistance
Varieties of Polymorphism

• **Parametric polymorphism** A single piece of code is typed generically
  – Imperative or first-class polymorphism
  – ML-style or let-polymorphism

• **Ad-hoc polymorphism** The same expression exhibit different behaviors when viewed in different types
  – Overloading
  – Multi-method dispatch
  – intentional polymorphism

• **Subtype polymorphism** A single term may have many types using the rule of subsumption allowing to selectively forget information
Summary

• Types are important in modern languages
  – Program organization and documentation
  – Prevent program errors
  – Provide important information to compiler

• Type inference
  – Determine best type for an expression, based on known information about symbols in the expression

• Polymorphism
  – Single algorithm (function) can have many types