

Compiler Construction

SMD163

Lecture 15: Polymorphism

Viktor Leijon & Peter Jonsson with slides by Johan Nordlander.
Contains material generously provided by Mark P. Jones



COMPUTER SCIENCE
AND ELECTRICAL ENGINEERING
LULEÅ UNIVERSITY OF TECHNOLOGY

1

Plan for This Lecture:

- ◆ A continuing look at type systems as an example of static analysis.
- ◆ How can we balance safety with flexibility?
 - One answer: Polymorphism
- ◆ How can we balance documentation with clutter?
 - One answer: Type Inference
- ◆ There's a lot we could talk about here, but we'll only be scraping the surface ...

2

Polymorphism:

- ◆ According to my dictionary:
polymorphism [poli *mawr* fizm] *n* occurrence of several types of individual organism within one species.
- ◆ From the Greek:
polymorphism = "many shapes".
- ◆ In programming languages:
A single value/function/object can be used at many different types.

3

Some Examples:

- ◆ Some operations work on only one type of value:
$$\frac{E \vdash e_1 : \text{boolean} \quad E \vdash e_2 : \text{boolean}}{E \vdash e_1 \ \&\& \ e_2 : \text{boolean}}$$
- ◆ Some operations work on any type of value:
$$\frac{E \vdash e_1 : T[] \quad E \vdash e_2 : \text{int}}{E \vdash e_1[e_2] : T}$$
- ◆ Some operations work only on some types of value:
$$\frac{E \vdash e_1 : t \quad E \vdash e_2 : t \quad t \in \{\text{int}, \text{float}\}}{E \vdash e_1 < e_2 : \text{boolean}}$$

4

Terminology:

- ◆ A monomorphic operator works on only one type of argument. (e.g., the && operator.)
- ◆ A polymorphic operator works on more than one type of argument.
 - Parametric polymorphism: essentially the same implementation/algorithm is used for all types of argument. (e.g., Array indexing.)
 - Ad-hoc polymorphism: different implementations are used for different types of value (e.g., numeric comparisons.)

5

Subtype Polymorphism:

- ◆ There is another kind of polymorphism in languages that allow implicit coercion of a value from one type to another.

$$\frac{E \vdash e : T \quad T < S}{E \vdash e : S}$$

- ◆ For example, in C, Java, etc., any `int` value can be used where a `float` is expected. (`int < float`)
- ◆ In Java, using the code for the quick calculator as an example, any `IntExpr` or `BinExpr` can be used where an `Expr` is expected. (`IntExpr < Expr`, `BinExpr < Expr`)

6

Implementing Polymorphism:

- ◆ Each different kind of polymorphism requires different implementation techniques, both for type checking and for run-time/execution.
- ◆ Each different kind of polymorphism is useful.
- ◆ Combining multiple forms of polymorphism in a single language can be challenging.
- ◆ We will be focusing on parametric polymorphism in this lecture.

7

Enforced Monomorphism:

- ◆ Suppose that `A` is a Java class, and consider the following function:

```
void swap(A[] arr, int x, int y) {
    A temp = arr[x];
    arr[x] = arr[y];
    arr[y] = temp;
}
```

- ◆ This definition will work for any choice of `A`.
- ◆ The definition uses only polymorphic constructs.
- ◆ But Java restricts it to a particular choice of `A`.

8

Lifting the Restriction:

- ◆ Let's introduce a way to indicate that A is a generic type – a parameter, not a fixed class:

```
void swap<A>(A[] arr, int x, int y) {
    A temp = arr[x];
    arr[x] = arr[y];
    arr[y] = temp;
}
```

- ◆ arr, x, and y are value parameters: any values will do (provided they have the right type!) ...
- ◆ A is a type parameter: any type will do ...

9

Using a Polymorphic Function:

- ◆ To use a polymorphic function, we just need to specify the type A as part of the call:

```
int[] intValues = new Int[] {1,2,3};
Button[] buttons = new Button[] {
    new
    Button("OK"),
    new
    Button("Cancel")
};
... swap<int>(intValues, 1, 2) ...
... swap<Button>(buttons, 0, 1) ...
```

10

Parameterized Types:

- ◆ Polymorphic functions start to become really useful in a language that has parameterized datatypes:

- Arrays of A's
- Pairs of A's and B's
- Lists of A's ...

11

Linked Lists:

- ◆ Linked lists are a very useful data structure:

```
class List {
    Value data;
    List next;
    List(Value data, List next) {
        this.data = data;
        this.next = next;
    }
}
```

- ◆ What's special about the Value type used here?
- ◆ ... Nothing! (i.e., any class would do)

12

A Profusion of Linked Lists:

- ◆ But we often need lots of different Value types in a single program:

```
class StringList {
    String data;
    StringList next;
    StringList(String data,StringList next) {
        this.data = data;
        this.next = next;
    }
}
class IntegerList { ... }
class ButtonList { ... }
```

- ◆ It's irritating to repeat this "boilerplate" over and over again ... and it makes the program harder to maintain ...

13

A Profusion of Functions too!

- ◆ We often need similar functions for each list type too:

```
int length(StringList list) {
    int len = 0;
    for (; list!=null; list=list.next) {
        len++;
    }
    return len;
}
int length(ButtonList list) {
    int len = 0;
    for (; list!=null; list=list.next) {
        len++;
    }
    return len;
}
```

Exactly the same code in each definition!

14

But why not use class Object?

- ◆ The class Object is the ultimate parent of all classes:

```
class List {
    Object data;
    List next;
    List(Object data, List next) {
        this.data = data;
        this.next = next;
    }
}
```

- ◆ Now we can easily build lists of Buttons as well as Strings...

15

But why not use class Object?

- ◆ However, this freedom actually applies to each particular list:

```
List x = new List(new String("abc"), null);
...
x = new List(new Button(), x);
...
```

- ◆ Moreover, all we know about the elements of a list is that they are "objects"; their type is Object. Thus, downcasts are needed:

```
String s = (String)(x.data);
```

- ◆ Now what if the data element of (the head of) x was accidentally a button? Such an error will not be caught until run-time!

16

A Parameterized List Type:

- ◆ In principle, we could avoid code duplication as well as downcasts by parameterizing the definition of the List class:

```
class List<Value> {
    Value data;
    List next;
    List(Value data, List<Value> next) {
        this.data = data;
        this.next = next;
    }
}
```

- ◆ This is an example of a parameterized datatype.

17

A Parameterized List Type:

- ◆ Type parameters now indicate the contents of each list:

```
List<String> x = new List<String>(new
    String("abc"),null);
...
List<Button> y = new List<Button>(new Button(), null);
...
```

- ◆ Lists of different type cannot be confused:

```
...
List<Button> y = new List<Buttons>(new Button(),x);
```

- ◆ Elements have their right type without casting:

```
String s = x.data;
Button b = y.data;
```

18

A Polymorphic Length Function:

- ◆ Moreover, we need only one definition for the length function:

```
int length<Value>(List<Value> list) {
    int len = 0;
    for (; list!=null; list=list.next) {
        len++;
    }
    return len;
}
```

- ◆ One concept, one definition.
- ◆ Less code to write, less code to understand, less code to maintain.

19

So how can this feature
be implemented?

20

Types:

- ◆ Previously, we have assumed we have a small collection of primitive types (int, boolean, ...), the array type constructor (int[], ...), as well as an extensible set of simple class types (Button, List, ...)
- ◆ Now we have added:
 - Parameterized types of the form $T\langle A_1, \dots, A_n \rangle$
 - Type parameters (or "type variables"), which are placeholders for arbitrary types that will be supplied later.

21

Type Checking Functions:

- ◆ Write $R(A_1, \dots, A_n)$ for the type of a function that returns a result of type R given arguments of type A_1, \dots, A_n .
- ◆ We will assume that an environment F is provided mapping function names to function types.
- ◆ Here is a rule for type checking a function call:

$$\frac{F(f) = R(A_1, \dots, A_n) \quad E, F \vdash e_1 : A_1 \quad \dots \quad E, F \vdash e_n : A_n}{E, F \vdash f(e_1, \dots, e_n) : R}$$

22

Continued ...

- ◆ Note that we have extended our typing rules to use hypotheses of the form $E, F \vdash e : A$ with two environments:
 - E, which records the names and types of variables;
 - F, which records the names and types of functions.
- ◆ We choose to keep these two environments separate because functions are not "first-class" values in our language. (They cannot be stored in data structures, or used as function parameters.)

23

Extending our Earlier Rules:

- ◆ Technically, all of the type checking rules that we have given previously need to be extended:

$$\frac{E, F \vdash e_1 : \text{boolean} \quad E, F \vdash e_2 : \text{boolean}}{E, F \vdash e_1 \ \&\& \ e_2 : \text{boolean}}$$

$$\frac{E, F \vdash e_1 : T[] \quad E, F \vdash e_2 : \text{int}}{E, F \vdash e_1[e_2] : T}$$

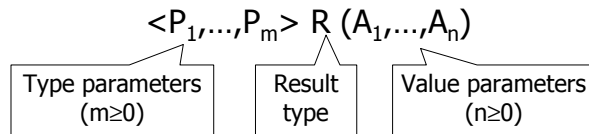
etc...

- ◆ None of these rules actually uses F ... but it might be needed in one of their hypotheses. This is why we need to propagate the new environment.

24

Checking Polymorphic Functions:

- ◆ To deal with polymorphic functions, we must extend our notation to allow function types of the form:



- ◆ For example, the length function has type: $\langle \text{Value} \rangle \text{int} (\text{List} \langle \text{Value} \rangle)$.

25

Instantiation:

- ◆ Polymorphic types are instantiated by picking a type to replace each of the parameters P_i .
- ◆ The instantiation of a polymorphic type $\langle P_1, \dots, P_m \rangle R (A_1, \dots, A_n)$, with types T_1, \dots, T_m :
 - Is written: $[T_1/P_1, \dots, T_m/P_m] R (A_1, \dots, A_n)$;
 - Is obtained by replacing each occurrence of a P_i in $R(A_1, \dots, A_n)$ with the corresponding T_i .
- ◆ We can instantiate the type of length to work on Strings. The result is $\text{int} (\text{List} \langle \text{String} \rangle)$.

26

Type Checking & Polymorphism:

- ◆ It is quite easy to extend our previous rule for type checking functions to work with polymorphic functions:

$$\begin{array}{c}
 F(f) = \langle P_1, \dots, P_m \rangle R (A_1, \dots, A_n) \\
 R' (A'_1, \dots, A'_n) = [T_1/P_1, \dots, T_m/P_m] R (A_1, \dots, A_n) \\
 \hline
 E, F \vdash e_1 : A'_1 \quad \dots \quad E, F \vdash e_n : A'_n \\
 \hline
 E, F \vdash f \langle T_1, \dots, T_m \rangle (e_1, \dots, e_n) : R'
 \end{array}$$

- ◆ This might look like a big mess of symbols ...
- ◆ Read carefully, and it will start to make sense!

27

The Hacker's Perspective:

To type check a call: $f \langle T_1, \dots, T_m \rangle (e_1, \dots, e_n)$

- ◆ Look up f in the function environment;
- ◆ Suppose that f has type $\langle P_1, \dots, P_m \rangle R (A_1, \dots, A_n)$;
- ◆ Replace each use of P_i in $R (A_1, \dots, A_n)$ with the corresponding T_i . Call the result $R' (A'_1, \dots, A'_n)$;
- ◆ Check that each argument e_i has type A'_i ;
- ◆ The result type of the call is just R' .

28

Implementing Types:

```
class Type {
    abstract boolean equal(Type other);
    ...
}
```

Test for equality of types
in the "obvious" way.

```
class ParamType extends Type {
    String name;
    Type[] params;
    ...
}
class TypeVar extends Type {
    String name;
    ...
}
```

29

Testing for Equality:

```
class ParamType extends Type { ...
    boolean equal(Type other) {
        if (other instanceof ParamType) {
            ParamType that = (ParamType)other;
            if (name.equal(that.name)
                && params.length ==
                that.params.length) {
                for (int i=0; i<params.length; i++) {
                    if
                    (!params[i].equal(that.params[i]))
                        return false;
                }
                return true;
            }
        }
        return false;
    }
}
```

30

Implementing Instantiation:

```
class Type {
    abstract Type inst(Type[] given, Type[] ps);
    ...
}
```

Instantiate a polymorphic type.

```
class TypeVar extends Type {
    String name;
    Type inst(Type[] given, Type[] ps) {
        for (int i=0; i<ps.length; i++) {
            if (this.equal(ps[i])) {
                return given[i];
            }
        }
        return this;
    }
}
```

$[T_i/P_i] P_i = T_i$

$[T_i/P_i] X = X$

31

Pragmatics:

- ◆ Although it is flexible, this type system can be rather cumbersome in practice:

```
List<Integer> state;
List<List<Integer>> states;
states = new
    List<List<Integer>>(state, states);
...
```

- ◆ Type names can quickly become very long, making them harder to write, and harder to get right.

- ◆ Can we get by without writing types?

32

Explicit vs Implicit Typing:

◆ In many languages, types are specified explicitly:

```
int f(int a, int b) {  
    int c = a + b;  
    int d = a - b;  
    return c*d;  
}
```

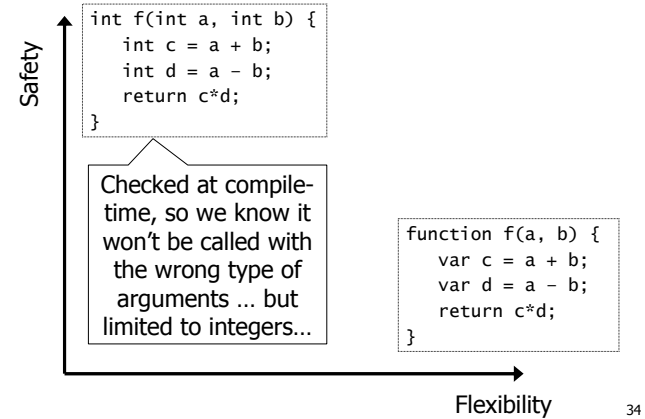
◆ Imagine what this might look like if types were implicit:

```
function f(a, b) {  
    var c = a + b;  
    var d = a - b;  
    return c*d;  
}
```

Valuable documentation ... or restrictive clutter?

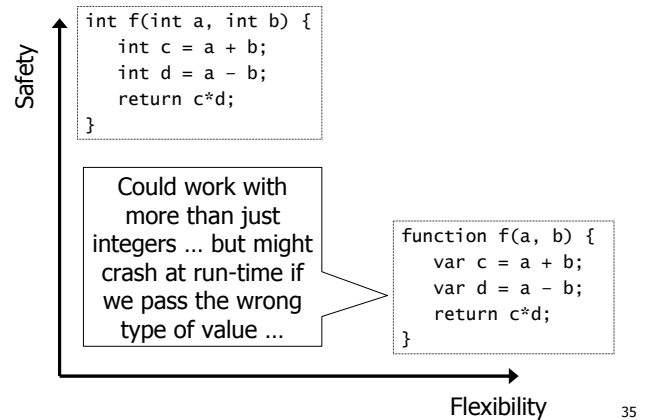
33

Typed and Safe:



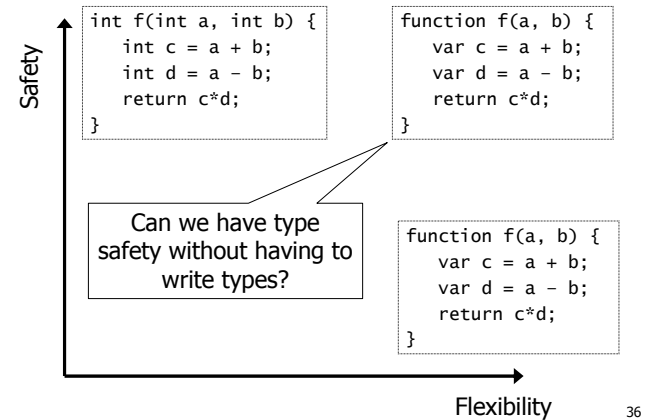
34

Untyped and Flexible:



35

Safety and Flexibility?



36

Type Inference:

- ◆ Take a language that:
 - Doesn't require you to declare the return type of each function;
 - Doesn't require you to declare the type of each function argument;
 - Doesn't require you to declare the type of each local variable.
- ◆ Try to infer the information that you need by looking at the context.
- ◆ In effect, get the compiler (static analysis) to add the type information, instead of the programmer.

37

For Example:

- ◆ Consider the following definition:

```
function choose(c, x, y) {  
    return (c ? x : y);  
}
```

- ◆ It's clear that *c* must be a boolean, and that *x* and *y* must have the same type.
- ◆ So we can treat *choose* as a polymorphic function:

```
A choose<A>(boolean c, A x, A y) {  
    return (c ? x : y);  
}
```

38

New Type Variables:

- ◆ The first time we encounter a variable like *x* or *y*, we won't know what type to use for it.
- ◆ So let's make up a "new" (or "fresh") type as a first guess ...
- ◆ ... but be prepared to adjust our guess as we see how the variable is actually used.

39

Type Checking choose:

When we first see the definition of *choose*, we know that it will have a type of the form $R(U,V,W)$ for some types *R*, *U*, *V*, *W*, but we don't know what those types will be ...

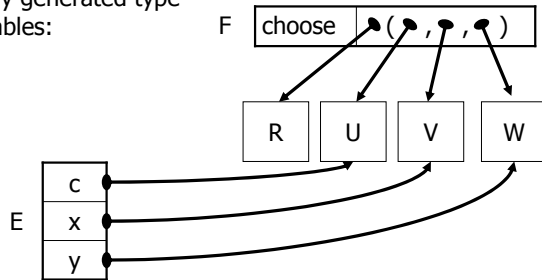
```
function choose(c, x, y) {  
    return c ? x : y;  
}
```

40

Type Checking choose:

So we will initialize our environments to use types that contain newly generated type variables:

```
function choose(c, x, y) {
  return c ? x : y;
}
```

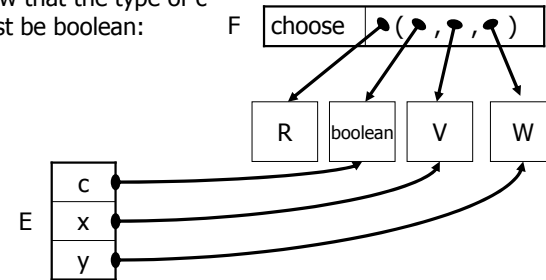


41

Type Checking choose:

Now we see c used as the first argument of a ?: expression, so we know that the type of c must be boolean:

```
function choose(c, x, y) {
  return c ? x : y;
}
```

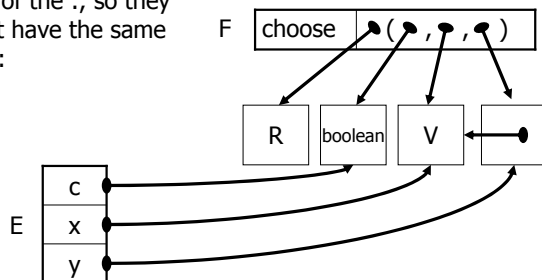


42

Type Checking choose:

Next we see that x and y are used as the two arguments on either side of the :, so they must have the same type:

```
function choose(c, x, y) {
  return c ? x : y;
}
```

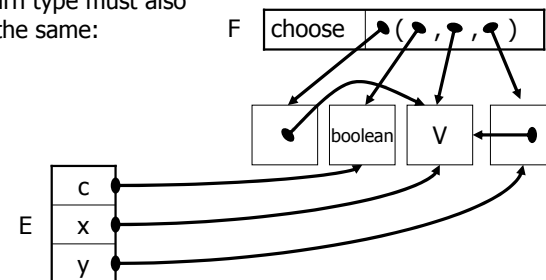


43

Type Checking choose:

Because this is the value that is returned from the function, the return type must also be the same:

```
function choose(c, x, y) {
  return c ? x : y;
}
```



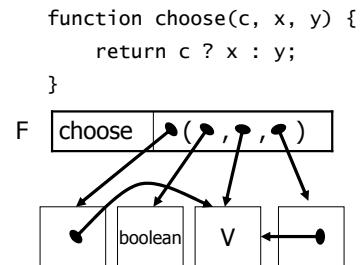
44

Type Checking choose:

Now we're done!

We leave the scope in which E was valid, and read off the inferred type for choose from F:

$V(\text{boolean}, V, V)$.

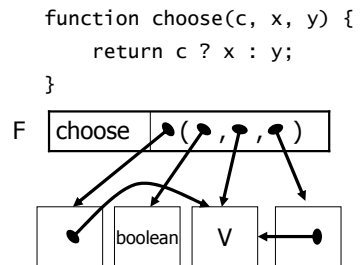


45

Type Checking choose:

Because V was a new variable, we can now infer that our function is polymorphic:

$\langle V \rangle V(\text{boolean}, V, V)$.



46

Implementing Type Variables:

```
class TypeVar extends Type {
    String name;

    static int nextNew = 0;
    static TypeVar fresh() {
        return new TypeVar("new" +
            nextNew++);
    }

    Type boundTo = null;
    ...
}
```

This is the code we use to make "fresh" types ...

We set this pointer to indicate when this type variable has been bound

47

Unification:

- ◆ It's no longer sufficient just to test two types for equality ...
- ◆ The types might point to distinct type variables, and hence look like distinct types.
- ◆ But we can make the two types equal by unifying the variables; that is, by arranging for one variable to point to the other.
- ◆ Thus, unification plays a central role in type inference.

48

Implementing Unification:

```
class TypeVar extends Type {  
    String name;
```

```
    Type boundTo = null;
```

```
    boolean unify(Type other) {  
        if (boundTo != null) {  
            return boundTo.unify(other);  
        } else if (!other.equal(this)) {  
            boundTo = other;  
        }  
        return true;  
    }  
}
```

What to do if this variable has already been bound ...

Otherwise we can force the two types to be equal by binding this variable.

49

What Happens at Runtime:

- ◆ How are polymorphic operations dealt with when a program is executed?
- ◆ There are two main strategies:
 - Heterogeneous;
 - Homogeneous.
- ◆ What are the main ideas? What are the tradeoffs?

50

The Heterogeneous Strategy:

- ◆ The compiler generates a different piece of code for each different use of a polymorphic function.
 - A glorified macro preprocessor?
 - Bigger programs ...
 - But at least the programmer doesn't have to maintain them all.
 - Separate compilation creates problems ... you won't know all of the ways that a polymorphic function will be used when you compile its definition.

51

The Homogeneous Strategy:

- ◆ Use *exactly* the same code for all versions of a polymorphic function.
 - Easier to implement, smaller programs.
 - A program may therefore manipulate data without knowing its type ...
 - ... but if you don't know its type, then you won't know how much memory it takes, and you won't be able to move it around ...
 - ... unless every single data item takes exactly the same amount of storage.
 - The need for such "uniform representations" can make this implementation strategy more expensive.

52

A Real Language:

- ◆ The syntax we've been using in this lecture is (almost) the syntax of a real language, GJ ("Generic Java", but, for legal reasons, they can't actually use that name ...).
- ◆ The syntax is horrible ... but that's because they didn't want to change the existing syntax of Java.
- ◆ Think "templates" in C++ ... (but better)
- ◆ Think "generics" in Ada ...
- ◆ Think typed functional languages ...
- ◆ This was implemented as "generics" in Java 1.5.
- ◆ The compiler in Sun's previous JDK release was actually the GJ compiler ... with the generic bits turned off! With Java 1.5 this changed...

53

Summary:

- ◆ Polymorphic type systems increase flexibility without compromising on safety.
- ◆ Type checking an explicitly typed polymorphic language is straightforward. (But the backend may need more work!)
- ◆ We can use type inference so that programmers don't have to write types explicitly.

54