4 Advanced typing

4.1 Classification of type systems

4.1.1 Existence & sophistication level

4.1.2 Orthogonality

4.1.3 Strong vs. weak typing

4.1.4 Statics vs. dynamic typing

4.1.5 Other kinds of typing

4.1.6 Type information responsibility

4.2 Structural

4.3 Theoretical polymorphism

4.3.1 Motivation

4.3.2 Overloading

4.3.3 Coercion

4.3.4 Universal polymorphism

4.3.5 Polytypes

4.3.6 Inclusion polymorphism

4.3.7 Summary

4.4 Polymorphism in practice

4.4.1 Overloading in PASCAL, C/C++, and JAVA

4.4.2 Coercion and the C++ overloading tournament

4.4.3 Polymorphic functions

4.4.4 Type inference

4.4.5 Checking parameters with parametric polymorphism

4.4.6 Case studies

4.5 References

4.6 Exercises

4.7 List of tables

4.8 List of figures

4.9 List of equations

References

Exercises

1. Criteria for the classification of type systems

Existence Does the language include a type system at all?

Sophistication level Assuming a PL has a type system, how rich is it?

Orthogonality Discriminatory vs. non-discriminatory

Strength How strictly are the typing rules enforced?

Time of Enforcement What stage is type checking performed? Static vs. dynamic typing

Responsibility Is the programmer responsible for type declarations or is it the compiler? Explicit vs. implicit typing

Equivalence When can one type replace another?

Flexibility To what extent does the type system restrict the user’s expressiveness? Polymorphic typing?

We will discuss criteria

1. Existence

2. Sophistication level

3. Orthogonality

4. Strength

5. Time of enforcement,

6. Responsibility

However,

7. Equivalence

8. Flexibility
deserve their own (fat) sections.

4.1.1 Existence & sophistication level

7 Frames: Existence of a type system? Example I: LISP is an untyped language Example II: MATHEMATICA is untyped as well Degenerate type systems Degenerate type systems in scripting languages Sophistication level of the type system Beyond types & non-standard types

A PL can be...

Typed The set of values can be broken into sets, with more or less uniform behavior under the same operation of the values in each set.

C, PASCAL, ML, ADA, JAVA, and most other PLs

Untyped Each value has its own unique set of permissible operations, and their semantics are particular to the value.

LISP, PROLOG, MATHEMATICA, ...
4. Example I: Lisp is an untyped language

In Lisp...

- All values are S-expressions
- An S-expression is an unlabeled binary trees, whose leaves are atoms
- Basic operations:
  - Car extracting left subtree
  - Cdr extracting right subtree
  - Cons construct a tree from two subtrees
- Null determine whether a tree is the atom NIL
- Short-hand notation:
  - cons(1, cons(2, cons(3, nil))) is shorthanded as (1 2 3).

Summary

Legality of operations is determined by tree structure and values at the leaves.

5. Example II: Mathematica is untyped as well

Mathematica is a language for symbolic mathematics:
- Values are symbolic mathematical expressions
- Expressions are trees
- All expressions have a head and body
  - head is a symbol
  - body is a list of expressions
- most details are not so interesting

The crucial point

Manipulation of an Expression: still requires
- check of legality
- determine of semantics

These are determined by
- tree structure of the expression
- values residing in internal nodes & leaves.

6. Degenerate type systems

Characteristics:
- Very few (typically one or two) primitive types
- Very few type constructors

Examples:
- BCPL (C’s ancestor) the only data type is a machine word
- DOS batch language the only data type is a string, which sometimes look like a file name
- Ancient Fortran: Several “full blown” primitive types for scientific computation
- Single type constructor: “array”

7. Degenerate type systems in scripting languages

AWK two data types

- strings
- numbers possible interpretation of some strings
  - only one “type constructor”:
    - associative array
    - No easy way to “create arrays of arrays”
  - most scripting PLs including BASH, and C-Shell are similar to AWK

8. Sophistication level of the type system

1. no typing
2. degenerate typing
3. non-recursive type systems
   - as in Fortran
4. recursive type systems
   - as in Pascal
5. functions as first class values
   - as in ML
6. highly advanced type constructors
  - “monads” of Haskell

9. Beyond types & non-standard types

Modern languages use the typing system to indicate more than just the type returned by a function.

Java the type of a method includes a list of the types of the exceptions that it can throw. Method M1 declares that it might throw exceptions of type E, and method M2 includes a call to M1, then either: The call is inside a try...catch(E) block, or M2 must also declare that it may throw E. The “catch-or-declare” principle.

C++ a const method cannot invoke a non-const method.

Haskell the type system indicates which functions perform I/O operations.

Shared theme: statically detect and prevent potential bugs.

4.1.2 Orthogonality

8 Frames: Orthogonality, discrimination & being second-class
- Discrimination vs. second-class types
- More on discrimination vs. second-class types
- Understanding orthogonality of a PL
- Orthogonality in Pascal
- First & second class values in Pascal
- Apparent contradiction?
- Value manipulation
10. Orthogonality, discrimination & being second-class

Definition 4.1.1 (Discriminatory type system). A type system is discriminatory if one of its type constructors is “discriminatory”, i.e., it is applicable to some types, but not to others.

In C++, the “reference to” type constructor is (self) discriminatory. There are types of references to almost everything. But, there are no references to references.

11. Discrimination vs. second-class types

Once upon a time, in the far far away kingdom called calligraphic capital $T^1$, lived a type named little $\tau$, and as we say in math, little $\tau \in T$. Little $\tau$ was looking for an employment by one of...

$C_1, \ldots, C_n \in T$ the type constructors of kingdom calligraphic capital $T$.

So, little $\tau$ went to type constructor capital $C_2$, but capital $C_2$ said No

So, little $\tau$ went to type constructor capital $C_7$, but capital $C_7$ also said No

and capital $C_3$ also said No

and capital $C_5$ said No

but, then capital $C_1$ said Yes

so, what did little $\tau$ do?...

12. More on discrimination vs. second-class types

Little $\tau$ was close to despair. She thought of capital $C_1$, and then of capital $C_2$, and, and, all the other $C_i$ who would not employ her, and then she realized that she was not a first class type, which means, ernyhh... that little $\tau$ must be second class type

Definition 4.1.2 (Second class types). if a type $\tau$ is being discriminated against by most (or even just very many) type constructors, then we tend to say that $\tau$ is a second class type.

Saying that $\tau$ is a second class type is cruel and unjust. It lets us refrain from accusing so many type constructors as being evil and discriminatory. But placing a “blame” on a second class types allows more compact expression and understanding of the language orthogonality

14. Orthogonality in Pascal

Non-discriminatory type constructor you can create arrays of anything,

Discriminatory type constructor you can create sets of Booleans and of Char, but not of Integer or of “set of Boolean”.

Second class type there are no “arrays of functions”, no “records of functions”, no “sets of functions”, no “pointers to functions”, etc.,

so we must “conclude” that functions are second class types.

15. First & second class values in Pascal

• First-class values
  Only simple, atomic values: truth values, characters, enumerands, integers, reals, and also pointers.

• Lower-class values
  can be passed as arguments, but cannot be stored, or returned, or used as components in other values
  – composite values (records, arrays, sets and files): cannot be returned!
  – procedure and function abstractions
  – references to variables (unless disguised as pointers)

16. Apparent contradiction?

Above we said, that in PASCAL:

• You can create arrays of anything!
• You cannot create arrays of functions!

Resolution:

• We like to think of the array constructor as being non-discriminatory
• We like to think of functions as second class
• We like to allow “non-discriminatory” constructors to discriminate against second class
• As we shall see, the “fault” lies with functions, not with arrays.

17. Value manipulation

• Operations on values
  – Passing them to procedures as arguments
  – Returning them through an argument of a procedure
  – Returning them as the result of a function
  – Assigning them into a variable
  – Using them to create a composite value
  – Creating/computing them by evaluating an expression
  – ...

• A value for which all these operations are allowed is called a first-class value
• We are used to integer or character values, but function values are also possible!

---

1 when we say “kingdom” we really mean “type system”
4.1.3 Strong vs. weak typing

4 Frames: □ Strong vs. weak typing □ Type punning □ Spectrum of strength □ Type punning in C#

18. Strong vs. weak typing

Two kinds of PLs

Strongly typed e.g., ML, EIFFEL, MODULA, JAVA

- it is impossible to break the association of a value with a type from within the framework of the language.
- it is impossible to subject a value to an operation which is not acceptable for its type.

Weakly typed e.g., assembly, C, C++, some variants of PASCAL

- values have associated types, but it is possible for the programmer to break or ignore this association.
- type punning...

in truth there is a spectrum of strength...

19. Type punning

“value is represented as a sequence of bits”, yet, types hide this fact.

Definition 4.1.3 (Type punning). type punning: revealing the mascaraed of bit sequences as values

Type punning has the power to Peep into the bit sequence implementation of a type Mutilate a value, by subjecting it to operations not allowed for its type

20. Spectrum of strength

Some of these languages are more safe than others

- PASCAL is more strongly typed than C; can still break type rules with:
  - Variant records
  - Through files

- JAVA is more strong typed than C#:
  - JAVA’s JVM guarantees (dynamically) strong typing
  - in C#, there are several ways of type punning

3See below, discussion of “structural typing”

21. Type punning in C#

In C#, type punning must be annotated with the unsafe keyword

```csharp
class Foo {
    public static void Main() {
        unsafe {
            // could also annotate class Foo
            int i = 14;
            int *p = &i;
            Console.WriteLine("I is \(i\) + \(i\));
            Console.WriteLine("I is also \(p->ToString()\));
            Console.WriteLine("It address in memory is \(int)p\);
        }
    }
}
```

4.1.4 Statics vs. dynamic typing

9 Frames: □ Type checking □ Time of enforcement □ Dynamic typing & type tags □ Cons of dynamic typing □ Characteristics of static typing □ Why static typing? □ Escaping the “evil” Dr. Rice □ Benefits of static typing □ Limits of static typing

22. Type checking

Definition 4.1.4 (Type checking). Language implementation applies type checking to ensure that no nonsensical operations occur.

Examples:

- **Multiplication** check that both operands are numeric.
- **Logical and** check that both operands are of Boolean type.
- **Field access** check that the operand is a Record containing the given field name.
- **Tuple access** check that the operand is a Tuple (array value) and that the index is valid.

23. Time of enforcement

- Type checking must be precede the operation
- could be done either at compile-time or at run-time:

Statically typed PLs

- type rules are enforced at compile time.
- every variable and every formal parameter have an associated type.

C, PASCAL, EIFFEL, ML, ...

Dynamically typed PLs

- type rules are enforced at run-time.
- variables, and more generally– expressions, have no associated type.
- only values have fixed types.

SMALLTALK, PROLOG, SNOBOL, APL, AWK, REXX,...
24. Dynamic typing & type tags

- Identifiers have no type associated with them.
- Types are associated with the values generated in run-time.
- Each value carries a type tag, identifying its type.

Pros

**Flexibility**  Arrays don’t have to be of a homogeneous type.
**Run partial programs**  An identifier needs to be “type-correct” only if accessed.
**Quick turnaround**  Faster development time.

25. Cons of dynamic typing

Conversely, “pros of static typing”

**Space overhead**  Each value is tagged with type information.
**Time overhead**  Tag must be examined at runtime.
**Unsafety**  Many type errors could have been detected by static compile-time checks.
**Obfuscation**  Entities annotated with type information are easier to understand.

yet, it seems as if the world is moving toward dynamically typed languages.

26. Characteristics of static typing

**Type**  annotation for each variable, parameter, function and procedure.
**Pre-Declaration**  usually means that all identifiers should be declared before used.
**Invariant of Values**  no value will ever be subject to operations it does not recognize.
**Invariant of Variables**  a variable may contain only values of its associated type.
**Invariant of Operations**  no operation, including user defined functions and procedures, will ever be applied to values of a type they do not expect.

27. Why static typing?

**Theorem (H. G. Rice)**

- Let \( f \) be any feature or property of the execution of computer programs.
- Suppose that \( f \) is not-trivial, i.e., that \( f \) holds for at least one program \( p_i \), and does not hold for at least some other program \( p_j \).
- Then, there is no general algorithm that, given a program \( p \), decides whether \( p \) exhibits feature \( f \) or not.

Examples:
- Cannot (systematically) decide if the program stops.
- Cannot (systematically) decide if the program is correct.
- Cannot (systematically) decide almost any other interesting run time property of a program.

28. Escaping the “evil” Dr. Rice

- We still need every little help in fighting the horrors of software development.
- Types manage to escape this “evil” theorem;

Several other automatic aids are:

**Garbage collection**  automatic memory management (run time)
**Const correctness**  no modification of const parameters (compile time)
**Design by contract**  assertions, invariants, preconditions and post-conditions: partial specification of a function (run time)
**Void safety**  to prevent null-pointer access
**Other**  JAVA makes every effort to ensure that an initialized variable is never used; compiler warnings, find bug heuristics,...

29. Benefits of static typing

Prevent run time crashes:
- Mismatch in # of parameters
- Mismatch in types of parameters
- Sending an object an inappropriate message
- Early error detection (supposedly) reduces development time

Enforce design decisions:
- Cost
- Effort
- More efficient and more compact object code
  - Values do not carry along the type tag
  - No need to conduct checks before each operation

30. Limits of static typing

- Some senseless operations cannot be statically checked.
- Generic examples:
  **Division by zero**  no typing system for the integers can prevent this without hitting the halting problem; checks are realized in run time by using machine-level exceptions. These checks carry no overhead.
  **Void safety**  Access to null pointers.
  **Other math issues**  square root of a negative value
  **Array references**  it is impossible to statically detect underflow or overflow of array indices.

4.1.5 Other kinds of typing

- Postmortem typing
- Mixed typing
- Mixed typing in JAVA
- Gradual typing
- Duck typing
- Notion of “type” with duck typing
31. Postmortem typing

Postmortem typing is just a different name for weak typing.

Definition 4.1.5 (Postmortem typing). Suppose that the execution of a type error is not detected by the PL’s runtime.

- Then, the program is allowed to merrily carry along with this error.
- In (hopefully) most cases, the program find its bitter end, shortly after.
- In other cases, the program will die much later or just behave strangely.

The posthumous analysis of errors is called “postmortem typing”.

The programmer is then responsible to remove the type errors from his program

Example: array overflow in C/C++.

32. Mixed typing

Definition 4.1.6 (Mixed typing). We say that a PL has mixed typing if the PL exercises both static typing and dynamic typing (usually for different purposes).

The PASCAL case

- Most checks are compile time
- Array bounds are checked at runtime
- Some type errors are not checked.

33. Mixed typing in Java

<table>
<thead>
<tr>
<th>Error</th>
<th>Compile time</th>
<th>Load time</th>
<th>Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>V = E</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>f(E)</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>null pointer access</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>array overflow</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1.1: Mixed typing in Java

34. Gradual typing

Some modern additions allow gradual introduction of types into your program

- Write your program as usual in a dynamically typed language.
- As the program matures, gradually add type annotation to:
  - Arguments to functions
  - Function return type

- The PL will cooperate behind the scenes:
  - Mark obvious type violations
  - Mark contradicting annotations
  - Reduce runtime overhead

35. Duck typing

Duck typing is a variant of “dynamic typing”

Given an operation op and a value v

Dynamic typing at run time

1. Determine type T for which op is defined.
2. Determine type T’ the type of v
3. If T’ ≤ T, execute op on v

Duck typing at run time

1. Determine O(v), the set of operations recognized by v, by either
   - determining T, the type of v
   - reading the list O(v) as attached to v
2. If op ∈ O(v), execute op on v

36. Notion of “type” with duck typing

Type of Values duck typing allows each value to have its own “type”.

Runtime type e.g.,

- each function
- each parameter
- each invocation

defines a set of operations that are being applied by

- this function
- to this parameter
- in this particular invocation.

Definition 4.1.7 (Duck typing error). A “duck” typing error occurs if a value’s type does not match the “runtime type” during the program runtime

4 Some PASCAL compilers have a flag for turning off array bounds checks
37. The 3 alternatives for declaration of types

I. Manifest typing as in C, Pascal

II. Inferred typing as in ML; AKA Implicit typing
most PLs can infer, at least partially, types. In ML it is particularly astonishing since it involves recursive functions and type parameters.

III. Semiimplicit typing
• Fortran: variables which begin with one of the six letters “i”, “j”, “k”, “l”, “m”, and “n” are integers; all others are real.  
• Basic (older versions): Suffixes such as % and $ determine the variable’s type.
• Perl: Essentially the same as Basic.

38. Manifest typing or explicit typing
Programmer is in charge for annotating identifiers of
• values
• variables
• functions
• procedures
• parameters
• ...
with type information.
found, e.g., in Pascal, Ada, C, Java, ...
Type annotation is also a documentation aid
X: speed; (* Good *)
Y: real; (* Bad *)
Z = 3; (* Worse *)

39. Responsibility for tagging
Definition 4.1.8 (Type inference). Compilers have the ability to apply type inference rules to e.g., to determine the type of expressions.

Why not apply this also to variables and other entities?
Definition 4.1.9 (Implicit typing). A programming language feature by which the programmer does not have to provide type information; the compiler infers the type of an entity from the way it is defined.

The risk of inadvertent creation of variables due to typos and spelling errors:
ML Answer no variables
Value declaration just like CONST declaration in Pascal

ML

ML Answer I careful analysis of the type system to detect when this problem may occur

ML Answer II type constraints

References
• Duck typing
• First class citizens
• First class functions
• Gradual typing
• Manifest typing
• Marshalling
• Name mangling
• Nominal typing
• Nominative & structural typing
• Serialization
• Structural vs. nominal typing
• Structural type System
• Type inference
• Type punning
• Type safety
• Type system

5The risk of inadvertent creation of variables can be precluded by the declaration implicit none
Exercises

1. Compare the terms “type error” and “type punning”.

2. • Write a program which never makes type errors, yet, would be rejected by a compiler implementing static typing.
• Did you know that
  
  Your program explains how static typing escapes Rice theorem.
• How is that?

3. “In PLs which exercise only static typing, values do not need to carry a type tag”.

   True or false? Explain.

4. Explain why the parallels between tuple and record values make us consider “index overflow” and “index underflow” as type errors?

5. Explain why “division by zero” is not usually regarded as type error.

6. What is the difference between “mixed typing” and “gradual typing”?

7. Explain why violating “void safety” is regarded as type error.

8. Why shouldn’t

   \[
   \arcsin(\sqrt{2})
   \]

   be marked as type error?

9. Explain the statement “reference types are closer to be first class citizens, in C++ than in Pascal” (Hint: where are the reference types in Pascal?).

10. In the above we saw an example in which the XOR function of two pointers.

    • Explain how this trick can be used for a compact representation of a bi-linked list, storing only one pointer in each node.

    • What limitations does this compact representation place on its users?

11. Explain why the combination of static typing and weak typing, as in the C PL, to never attach type tags to values.

4.2 Structural vs. nominal typing

42. Type equivalence

• Suppose that an operation expects an operand of type \(T\), but receives an operand of type \(T'\). Is this an error?

• No, if \(T\) is a subtype of \(T'\)

• Two types that are subtypes of each other are called equivalent

• Caveat: because the notion of subtype is more refined than the notion of type equivalence, we will find languages where type equivalence is defined but subtyping is not

• Kinds of type equivalence:
  
  – Structural equivalence
  – Name equivalence
  – Declaration equivalence

43. Structural equivalence

• Have the “same” values... In Algol-68: if \(T\) and \(T'\) are both atomic, then \(T = T'\) if, and only if, they are identical. Else if

  \[
  - T = A \times B, T' = A' \times B', \text{ or}
  - T = A + B, T' = A' + B', \text{ or} \quad T = A \rightarrow B, T' = A' \rightarrow B' \text{ and } A = A', B = B'
  \]

  Otherwise? Recursive types:

  \[
  - T = \text{Unit} + c(A \times T)
  - S = \text{Unit} + c(A \times S)
  - R = \text{Unit} + c(A \times S)
  \]

  intuitively \(T\), \(S\) and even \(R\) are structurally equivalent, but structural equivalence is not easy to define and test for in recursive types.

• Points of disagreement between languages:

  – Do records require field name identity to be structurally equivalent?

  – Do arrays require index range identity to be structurally equivalent?

44. An example for (hypothetical) structural equivalence

45. Name equivalence

if, and only if, \(T\) and \(T'\) were defined in the same place (original Pascal and Ada):
46. Name equivalence across programs

Pascal

Program p1(f)
TYPE
  T = file of Integer;
VAR
  f: T;
Begin
  ...
  Write(f,...);
  ...
end;

Pascal

Program p2(f)
TYPE
  T = file of Integer;
VAR
  f: T;
Begin
  ...
  Read(f,...); (* X *)
  ...
end;

By definition of Pascal, it follows that

- Two Pascal programs cannot communicate legally through files using any user-defined type, hence the type error above.
- However, multiple instances of the same program can.
  - type Text, which is the only predefined type for files in Pascal, allows communication between programs.
- In practice, most implementations of Pascal do not type check files. Thus, these implementations subvert Pascal’s type safety, but allow reasonable interfacing.

47. Name equivalence across programs

48. Declaration equivalence

- A later Pascal standard (ANSI 1983): \( T = T' \) if, and only if, \( T \) and \( T' \) have the same declaration
- Problem across programs: gone.
- Sub-typing in Pascal
  - A weaker notion than type equivalence: an operation that expects an operand of type \( T \) but receives an operand of type \( T' \) is legal also if \( T \) and \( T' \) are not equivalent, but \( T \) is a subtype of \( T' \).
- Sub typing in Pascal- only in one case:
  - if \( T = [a..b] \) and \( T = [c..d] \) then \( T \) is a subtype of \( T' \) if, and only if, \( T \) is a subrange of \( T' \), i.e.,
    \[
    a \leq c \leq d \leq b
    \]
  - Limitation: there is no way to override this definition.

49. Derived types in Ada

- Subtypes in Ada More about Sub-typing in Ada
- Sub-typing of ordered primitive types - using subranges:

50. Type equivalence

- Suppose that an operation expects an operand of type \( T \), but receives an operand of type \( T' \). Is this an error?
  - No, if \( T' \) is a subtype of \( T' \).
- Two types that are subtypes of each other are called equivalent Caveat because the notion of subtype is more refined than the notion of type equivalence, we will find languages where type equivalence is defined but subtyping is not
- Kinds of type equivalence:
  - Structural equivalence
  - Name equivalence
  - Declaration equivalence

51. Structural equivalence

Have the “same” values In Algol-68:

- If \( T \) and \( T' \) are both primitive, then \( T \cong T' \) only if they are identical.
- Else if
  - \( T = A \times B, T' = A' \times B' \), or
  - \( T = A + B, T' = A' + B' \), or \( T = A + B, T' = B' + A' \), or
  - \( T = A \rightarrow B, T' = A' \rightarrow B' \)
and \( A \cong A', B \cong B' \Rightarrow T \cong T' \)
- Otherwise?

Recursive types:

- \( T = Unit + c(A \times T) \)
- \( S = Unit + c(A \times S) \)
- \( R = Unit + c(A \times S) \)

Intuitively \( T, S \) and even \( R \) are structurally equivalent, but structural equivalence is not easy to define and test for in recursive types.

52. Points of disagreement between languages

- Do records require field name identity to be structurally equivalent?
- Do arrays require index range identity to be structurally equivalent?

53. Name equivalence

\( T \cong T' \) if, and only if, \( T \) and \( T' \) were defined in the same place (original Pascal and Ada):

Program p1(f)
TYPE
  T = File of Integer;
VAR
  f1: T1;
  f2: T2;
Procedure p(VAR f: T1);
  ...
  p(f1); (* X *)
  p(f2); (* X *)

Program p2(f)
TYPE
  T = File of Integer;
VAR
  f: T;
Begin
  ...
  Read(f,...); (* X *)
  ...
end;
54. Name equivalence across programs

- By definition of PASCAL, it follows that
  - two PASCAL programs cannot communicate legally through files using any user-defined type, hence the type error above.
  - However, multiple instances of the same program can.
  - type Text, which is the only predefined type for files in PASCAL, allows communication between programs.
- In practice, most implementations of PASCAL do not type check files. Thus, these implementations subvert PASCAL’s type safety, but allow reasonable interfacing.

55. Declaration equivalence

- A later PASCAL standard (ANSI 1983): \( T \equiv T' \) only if \( T \) and \( T' \) have the same declaration

```
TYPE T1 = File of Integer;
T2 = File of Integer;
VAR f1: T1;
f2: T2;
Procedure p(Var f: T1);
p(f1); (* X *)
p(f2); (* X *)
```

- Problem across programs: gone.

4.3 Theoretical polymorphism

Contents [64 frames]

<table>
<thead>
<tr>
<th>Section</th>
<th>Frames</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3.1</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>4.3.2</td>
<td>18</td>
<td>12</td>
</tr>
<tr>
<td>4.3.3</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>4.3.4</td>
<td>5</td>
<td>16</td>
</tr>
<tr>
<td>4.3.5</td>
<td>11</td>
<td>17</td>
</tr>
<tr>
<td>4.3.6</td>
<td>10</td>
<td>18</td>
</tr>
<tr>
<td>4.3.7</td>
<td>1</td>
<td>20</td>
</tr>
<tr>
<td>References</td>
<td>1 frame</td>
<td>20</td>
</tr>
<tr>
<td>Exercises</td>
<td></td>
<td>20</td>
</tr>
</tbody>
</table>

4.3.1 Motivation

12 Frames: 
- The road to polymorphism
- Benefits of strong static typing
- An annoying PASCAL example
- Flexibility of type system
- Life without handcuffs can be wonderful!
- Responses to inflexibility
- Monomorphic vs. polymorphic type systems
- What’s monomorphism?
- Monomorphism of user defined functions in PASCAL
- Monomorphism of user defined functions in C
- “More than one type” ≈ polymorphism
- Ad hoc polymorphism ≠ polymorphic type system

56. The road to polymorphism

Figure 4.3.1: The road to polymorphism

57. Benefits of strong static typing

Large software systems tend to use static strongly typed languages, because of

Safety fewer bugs

Efficiency fewer runtime checks, and more efficient use of memory

Clarity typing makes the code clearer

However, typing can be a nuisance: the utility of a given piece of code may be very restricted by typing.

58. An annoying Pascal example

```
Procedure sort(var a: Array[1..300] of T);
```

could not be applied to

- Arrays of real (body and declaration has to be repeated with T=Real).
- \( \text{array}[1..299] \) of T: Array is too small.
- \( \text{array}[1..500] \) of T: Array is too large.
- \( \text{array}[0..299] \) of T: Mismatch of indices.
- \( \text{array}[1..300] \) of T: No name equivalence!!!!

PASCAL is so fussy and inflexible in its type system that even two identical type declarations are considered distinct. A type declaration made at a certain point in a program is equivalent only to itself.
59. Flexibility of type system

Flexible type system makes typing an aide, not a hurdle

- Avoid issuing type error messages on programs which will not make run type errors.
- Promotes code reuse for many different types.

Clearly, PASCAL offers a very inflexibility type system.

The holy grail of language design

Simultaneously maintain:

1. Flexibility
2. Safety
3. Simplicity

60. Life without handcuffs can be wonderful!

In dynamically typed Languages, polymorphic code may be invoked with variables of different type (writing almost at a pseudo-code level)

```plaintext
search(k) {
    // k is the key to search for
    // p is the current position in the search for k
    for (p = first(); not exhausted(p,k); p = next(p,k))
        if (found(p,k))
            return true;
    return false;
}
```

Alas

Very flexible, but not so safe

61. Responses to inflexibility

1. The C camp: Weak typing.

```plaintext
int qsort(
    char *base, //Start of array
    int n, // Number of elements
    int width, // Element’s size
    int (*)(cmp)()); //How elements are compared
);
```

2. Dynamically typed languages camp: SMALLTALK, PYTHON, etc.: dynamic typing overcomes complex inflexibility problems. In a sense, all code is polymorphic.

3. ADA/C++/JAVA camp: Polymorphic type systems

```plaintext
generic
type T is private
    with function comp(x: T, y: T)
    procedure sort(a: array(1..max) of T)
    procedure int_sort is new sort(int , "c");
...
```

But, what is a “polymorphic type system”?

---

59. Flexibility of type system

Flexible type system makes typing an aide, not a hurdle

- Avoid issuing type error messages on programs which will not make run time type errors.
- Promotes code reuse for many different types.

Clearly, PASCAL offers a very inflexibility type system.

The holy grail of language design

Simultaneously maintain:

1. Flexibility
2. Safety
3. Simplicity

60. Life without handcuffs can be wonderful!

In dynamically typed Languages, polymorphic code may be invoked with variables of different type (writing almost at a pseudo-code level)

```plaintext
search(k) {
    // k is the key to search for
    // p is the current position in the search for k
    for (p = first(); not exhausted(p,k); p = next(p,k))
        if (found(p,k))
            return true;
    return false;
}
```

Alas

Very flexible, but not so safe

61. Responses to inflexibility

1. The C camp: Weak typing.

```plaintext
int qsort(
    char *base, //Start of array
    int n, // Number of elements
    int width, // Element’s size
    int (*)(cmp)()); //How elements are compared
);
```

2. Dynamically typed languages camp: SMALLTALK, PYTHON, etc.: dynamic typing overcomes complex inflexibility problems. In a sense, all code is polymorphic.

3. ADA/C++/JAVA camp: Polymorphic type systems

```plaintext
generic
type T is private
    with function comp(x: T, y: T)
    procedure sort(a: array(1..max) of T)
    procedure int_sort is new sort(int , "c");
...
```

But, what is a “polymorphic type system”?

---

62. Monomorphic vs. polymorphic type systems

Monomorphic Type Systems

- Used in classical PLs, e.g., PASCAL
- Every entity has a single simple type
- Type checking is straightforward
- Unsatisfactory for reusable software;
  - Many standard algorithms are inherently generic (e.g., sort)
  - Many standard data structures are also generic (e.g., trees)

Polymorphic Type Systems

- Appear in modern languages, e.g., ADA, C++, JAVA and ML.
- Entities may have multiple types
- Code reuse thanks to universal polymorphism
- Supports
  - Generic functions, e.g., sort.
  - Generic types, e.g., binary tree.

63. What’s monomorphism?

In a monomorphic type system, functions (and other entities) have one, and only one, type.

```plaintext
Monomorphic = “single-shaped”
```

f is a function ⇒ |types(f)| = 1. (4.3.1)

64. Monomorphism of user defined functions in Pascal

Programmer defined functions (and procedures) in PASCAL are monomorphic:

```plaintext
Function gcd(a, m: Integer): Integer;
Begin
    if n mod m <> 0 then
gen := gcd(m, n mod m)
    else
gen := m;
End;
```

Function gcd is monomorphic:

```
|types(gcd)| = |{Integer × Integer → Integer}| = 1
(4.3.2)
```
65. Monomorphism of user defined functions in C

Function gcd is monomorphic:

\[ \text{types}(\text{gcd}) = \left\{ \text{int} \times \text{int} \rightarrow \text{int} \right\} = 1 \]  
(4.3.3)

66. “More than one type” \(\approx\) polymorphism

Poly-Morphism = poly + morphos [Greek] = many + form.

literally, the capacity of an entity to have several shapes

67. Ad hoc polymorphism \(\neq\) polymorphic type system

Overloading minimal utility. A (small) number of distinct procedures that just happen to have the same identifier.

- Not a truly polymorphic object
- Does not increase the language’s expressive power
- Similarity between shapes is coincidental

Coercion a little greater utility

- Same routine can be used for several purposes
- Number of purposes is limited
- Return type is always the same
- Similarity between shapes is determined by coercion operations which are external to the routine

4.3.2 Overloading

Definition 4.3.1 (Overloading). An overloaded term is a term that has multiple meanings, which may, but also may not be related.

How did the Marx trick work?

- Overloading
- Unrelated meanings
- Misleading context:
  (On its own, the phrase “Fruit flies like a banana” is not so confusing)

71. Overloading in English

Unrelated meanings:

lie to present false information with the intention of deceiving

“I did not lie in the deposition”

lie to place oneself at rest in a horizontal position

“I did not lie in this position”

Close (more or less) meanings:

fly to move through the air
fly to travel by an airplane
fly a two winged insect, such as insect

**Lemma 4.3.2** (The fundamental rule of overloading). The intended meaning is figured out by context.

72. Context dependent resolution of overloading

**Figure 4.3.5:** Archie Bunker explains to Edith Bunker how context is used to resolve the ambiguity of the three overloaded meanings of the word “Shalom” in Hebrew.

1973

http://tinyurl.com/overloading

Paul: Shalom.
Edith Bunker: Shalom? What does that mean?
Mike Stivic: Believe it or not, Ma, it means “peace”.
Gloria Stivic: Jewish people also use it to say “hello” and “good-bye”.
Edith Bunker: How do you tell if they mean “hello” or “good-bye”?
Archie Bunker: Simple, Edith. If a Jew is walking towards you, it means “hello”. If he’s walkin’ away, it means “good-bye”.
Edith Bunker: When does it mean “peace”? Archie Bunker: In between “hello” and “good-bye”.

73. Keyword overloading in Pascal

Keyword “of” serves several similar meanings in different contexts:

<table>
<thead>
<tr>
<th>Arrays</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CONST</strong></td>
</tr>
<tr>
<td>N = 100;</td>
</tr>
<tr>
<td><strong>TYPE</strong></td>
</tr>
<tr>
<td>Range = 1..N;</td>
</tr>
<tr>
<td>Matrix = Array[Range] of Real;</td>
</tr>
</tbody>
</table>

74. “Plain” overloading vs. identifier/-operator overloading

- Overloading of **of** in Pascal is not of an **identifier**.
- Keyword **of** does not identify an entity such as variable, a function, a procedure, etc.

**Definition 4.3.3** (Identifier overloading). An **identifier** or **operator** is said to be overloaded if it simultaneously denotes two or more distinct functions or procedures.

Operator “+” in Pascal and C denotes two distinct functions:

- Integer addition
- Floating point addition

Identifiers may be similarly overloaded...

75. Use of overloading for type polymorphism

We say that a function (or, an operator) f is overloaded, if:

- f has more than one type
- there is no automatic mechanism that generates the set types(f)

Overloading provides a mechanism for better utilization of scarce “good” names.
76. Builtin procedure overloading in Pascal

Program Write;
TYPE
  Days = (Sunday, Monday);
Begin
  WriteLn(0);
  WriteLn(0.0);
  WriteLn(false);
  WriteLn(Sunday)
  end.

Output is
0 0.000000000E+00 FALSE Sunday

The identifier WriteLn in PASCAL denotes many distinct functions.

77. Builtin function overloading in Pascal

Many PASCAL functions apply to more than one type:
- eof
- succ
- ord
- sin

But, their polymorphism is not overloading; it is either
- coercion, or
- parametric polymorphism.

Similarity of overloaded meanings is a matter of coincidence

78. Overloaded builtin operators of Pascal

Consider, e.g., operator +

Number of types More than one type:
- + ∈ Integer → Integer
- + ∈ Real → Real
- + ∈ Integer × Integer → Integer
- + ∈ Real × Real → Real

Thus, |types(+)| = 4 > 1.

Regularity Is the set of types “automatically generates?"
- The above four types are similar and related.
- They were designed by the individual who designed PASCAL.
- This individual gave them semantics.
- Incidentally, these semantics are related.
- But, they were not “automatically” generated.

79. Programmer defined overloading

- C does not allow operator overloading by programmer.
  but, its younger, fatter, and uglier, “daughter”, C++, does
- PASCAL does not allow operator overloading by programmer.
  but, its younger, fatter, and uglier, “sister”, C++, does

80. Function overloading in C++

C forbids function overloading, but its young, fat, and ugly sister, C++, welcomes it:

Function overloading in C++

```c++
double max(double d1, double d2) {
  return d1 > d2 ? d1 : d2;
}
char max(char c1, char c2) {
  return c1 > c2 ? c1 : c2;
}
char* max(char* s1, char* s2) {
  return strcmp(s1, s2) > 0 ? s1 : s2;
}
const char* max(const char* s1, const char* s2) {
  return strcmp(s1, s2) > 0 ? s1 : s2;
}
```

Neither C, nor C++ have “builtin” functions. Hence, they have no builtin function overloading.

81. Overloading the division operator in Ada

- PASCAL forbids operator overloading by programmer,
  but, its younger, fatter, and uglier, “daughter”, Ada, allows it:
- Builtin semantics of “/”:
  
  Integer division Integer × Integer → Integer
  Real division Real × Real → Real

Programmer defined overloading:

```ada
function "/" (m, n : Integer) return Float is
begin
  return Float(m) / Float(n);
end;
```

Adds another meaning to division of integers: it can now also return a real number.

82. Resolving ambiguity of overloading

The actual meaning is determined by context:
(i) which parameters are passed to operator “/” upon invocation
(ii) how its result is used.

83. Ambiguity resolution

Consider the call Id(E) where Id denotes both:
- a function f₁ of type S₁ → T₁
- a function f₂ of type S₂ → T₂

Context Independent (C++)
- Either f₁ or f₂ is selected depending solely on the type of E
We must have $S_1 \neq S_2$
May lead to ambiguities in the presence of coercion
Context Dependent (ADA)
Either $f_1$ or $f_2$ is selected depending on both on the type of $E$ or how $1d(E)$ is used.
Either $S_1 \neq S_2$ or $T_1 \neq T_2$ (or both).
Ambiguity is not always resolved:

$$x : \text{Float} := (7/2)/(5/2);$$
Has at least two ambiguous interpretations:
- $3/2 = 1.5$
- $3.5/2.5 = 1.4.$

### 84. Overloading vs. hiding

Hiding (by lexical scope): an identifier defined in an inner scope hides an identifier defined in an outer scope.

#### Hiding in C

```
static long tail;
int main(int ac, char **av) {
    // hides outer tail
    const char **tail = av + ac - 1;
    ...
```

Comparison: both do not make polymorphic types.

**Overloading** Multiple meanings co-exist

**Hiding** New meaning masks the old meaning.

### 85. Overloading & hiding together?

May be challenging for language designers?
- Can inner definition overload external definition?
- What happens if an inner definition hides one overloaded outer definition, but not the other?

#### Exercise

Provide examples in concrete languages, and see how they deal with these dilemmas.

### 4.3.3 Coercion

7 Frames: □ What's coercion? □ Why coercion? □ Implicit use of coercion □ Type coercion in ALGOL-68 □ Built-in coercion in C++ □ Ambiguity due to coercion □ Coercion + overloading

#### 86. What’s coercion?

**Typing**

- Polymorphic typing
- Monomorphic typing
- Ad hoc
- Universal
- Inclusion
- Parametric
- Overloading
  - Coercion
- Subrange
- Inheritance
- Polytypes

**Definition 4.3.4 (Coercion).** Coercion is a conversion from values of one type to values of another type which occurs implicitly, casting.

PASCAL provides coercion from **Integer** to **Real**, so we can write:

```
Function isPrime(n: integer): Boolean;
    VAR
    d: Integer; (* Potential divisor *)
    primeSoFar: Boolean;
    Begin
        If n < 0 then n := -n;
        primeSoFar := n >= 2;
        d := 2;
        While primeSoFar and (d <= sqrt(n)) do
            Begin
                primeSoFar := n mod d <> 0;
                d := d + 1;
            end;
        isPrime := primeSoFar;
    end;
```

- Function $\sqrt{n}$ expects a **Real**, but we need to compute $\sqrt{n}$, so we can
  write:

```
Function sqrt(n: integer): Real;
    VAR
    d: Integer; (* Potential divisor *)
    primeSoFar: Boolean;
    Begin
        If n < 0 then n := -n;
        primeSoFar := n >= 2;
        d := 2;
        While primeSoFar and (d <= sqrt(n)) do
            Begin
                primeSoFar := n mod d <> 0;
                d := d + 1;
            end;
        return sqrt(n);
    end;
```

1. Function $\sqrt{n}$ expects a **Real**, but thanks to coercion we can pass it an **Integer**.
2. Function $\sqrt{n}$ returns a **Real**, but thanks to coercion we can compare its result with an **Integer**

#### 88. Implicit use of coercion

Coercion enhances the utility of existing functions.

```
while primeSoFar and (d <= sqrt(n)) do ...
```

#### 89. Type coercion in Algol-68

ALGOL 68 allows the following coercion operations:

**Promotion** From integer to real

**Widening** From real to complex number

**Dereferencing** From reference to a variable to its value

**Rowing** From any value to a singled value array and more...

Now you can understand why modern languages tend to minimize or even eliminate coercion altogether.

#### 90. Built-in coercion in C++

```
int pi = 3.14159; // Built-in coercion from double to int
float x = 'A'; // Built-in coercion from char to float
extern double sqrt(float);
x = sqrt(pi); // Built-in coercion from int to double
// and then
// Built-in coercion from double to float
```

Coercion is sometimes called, especially in C++, **type casting** and **type conversion**, without particular distinction between implicit and explicit applications.
91. Ambiguity due to coercion

Graph of coercion operations is not always a tree

- What is the path of coercion from unsigned char to long double?

unsigned char → char → int → long → double → long double

or maybe,

unsigned char → unsigned → unsigned long → long double

- Selecting a different path may lead to slightly different semantics
- K&R C, ANSI-C and C++ are all different in this respect.

Graph of coercion operations is not always a DAG

- Types int, double and float in C, can all be coerced into each other.

- Therefore, the language definition must specify exactly the semantics of e.g., 'a' * 35 + 5.3f

92. Coercion + overloading

Strategies for support of mixed type arithmetic, e.g., A + B

Overloading and no coercion

- integer + integer
- real + integer
- integer + real
- real + real

Coercion and no overloading

- real + real
- integer → real

Coercion and overloading

- integer + integer
- real + real
- integer → real

4.3.4 Universal polymorphism

5 Frames: □ What is “ad hoc” polymorphism? □ Ad hoc vs. universal polymorphism □ The benefits of universal polymorphism □ Annoying example: a monomorphic Pascal function □ Using the disjoint monomorphic function

93. What is “ad hoc” polymorphism?

- Each overloaded version
- Each distinct coercion

Human can be language designer and/or programmer (depending on the PL)

94. Ad hoc vs. universal polymorphism

Definition 4.3.5 (“ad hoc”). Ad hoc adv. 1. For the specific purpose, case, or situation at hand and for no other: a committee formed ad hoc to address the issue of salaries. —ad hoc adj. 1. Formed for or concerned with one specific purpose: an ad hoc compensation committee. 2. Improvised and often impromptu: “On an ad hoc basis, Congress has ... placed ... ceilings on military aid to specific countries” (New York Times). [Latin ad, to + hoc, this.]

<table>
<thead>
<tr>
<th>Universal</th>
<th>Ad Hoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>No. Shapes</td>
<td>Unbounded</td>
</tr>
<tr>
<td>Shape Generation</td>
<td>Automatic</td>
</tr>
<tr>
<td>Shape Uniformity</td>
<td>Systematic</td>
</tr>
</tbody>
</table>

Table 4.3.2: Ad hoc vs. universal polymorphism

95. The benefits of universal polymorphism

- A single function (or type) has a (large) family of related types
- The function operates uniformly on its arguments, whatever their type.
- Provide a genuine gain in expressive power, since a polymorphic function may take arguments of an unlimited variety of types

96. Annoying example: a monomorphic Pascal function

Determine whether two sets of characters are disjoint

Type CharSet = set of Char;
Function disjoint(s1, s2: CharSet): Boolean;
Begin
  disjoint := (s1 * s2 = [])
End

Applicable only to sets of Char.

Type is $\wp(\text{Char}) \times \wp(\text{Char}) \rightarrow \text{Boolean}$

Applicable to a pair of arguments, each of type $\wp$Char:

VAR chars : CharSet;
Begin
  ... If disjoint(chars, ['a', 'e', 'i', 'o', 'u']) then ...;
End

All we have seen so far is ad hoc polymorphism, in which the variety in different shapes is created by human.
But, cannot be applied to arguments of other type, such as, \( \phi \text{Integer}, \phi \text{Color}, \ldots \)

Counter example: a Pascal polymorphic operator
The \( * \) operator in Pascal is polymorphic. It can be applied to any two sets of the same kind of elements.

Polymorphism is universal, since the operator works in the same fashion for all types for which it is applicable.

### 4.3.5 Polymtypes

#### 98. What are polytypes?

- **Typing**
  - Polymorphic typing
  - Monomorphic typing

- **Definitions**
  - **Polyp type** (also called parametric type) a whose definition contains one or more type variables
  - **Monotype** a type whose definition includes no type variables;
  - **Monomorphic PL** offers solely monotypes
  - **Polymorphic PL** offers also polytypes

#### 99. Polymtypes

- **Examples**: A “plain” polytype, and plenty of types of polymorphic functions:
  - \( \text{list}(\sigma) \)
  - \( \text{list}(\sigma) \rightarrow \sigma \)
  - \( \text{list}(\sigma) \rightarrow \text{Integer} \)
  - \( \sigma \rightarrow \sigma \)
  - \( \sigma \times \sigma \rightarrow \sigma \)
  - \( (\beta \rightarrow \gamma) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma) \)

#### 100. A polytype derives many types

A polytype derives a whole family of types, e.g., type \( \sigma \rightarrow \sigma \) derives:

- \( \text{Integer} \rightarrow \text{Integer} \)
- \( \text{String} \rightarrow \text{String} \)
- \( \text{list(Real)} \rightarrow \text{list(Real)} \)
- ...

### 101. No programmer-defined polytypes in Pascal!

The type of the predefined function \( \text{eof} \) is \( \text{File of } \sigma \). If Pascal had user defined polytypes, we could have written

- **TYPE**
  - \( \text{Pair}(\sigma) = \text{Record} \)
  - First, second: \( \sigma \);
  - end;
  - \( \text{IntPair} = \text{Pair}(\text{Integer}); \)

Unfortunately, this would not work in Pascal. All we can write is something of the sort of

- **TYPE**
  - \( \text{IntPair} = \text{Record} \)
  - First, second: \( \text{Integer} \);
  - end;
  - \( \text{line: CharList}; \)

- **Notations for some common polytypes:**
  - \( \text{Pair}(\sigma) = \sigma \times \sigma; \)
  - \( \text{datatyp} \sigma \text{ list} = \)
  - \( \text{cons of } (\sigma \times \sigma \text{ list}); \)
  - \( \text{fun hd}(l: \sigma \text{ list}) = \)
  - \( \text{case } l \text{ of nil } \Rightarrow \ldots \) (* error *)
  - \( \text{and tl}(l: \sigma \text{ list}) = \)
  - \( \text{case } l \text{ of nil } \Rightarrow \) (* error *)
  - \( \text{and length}(l: \sigma \text{ list}) = \)
  - \( \text{case } l \text{ of nil } \Rightarrow 0 \)
  - \( \text{and cons(h,t) } \Rightarrow 1 \) + \( \text{length } (t) \)

### 102. Defining polytypes in ML

- **TYPE**
  - \( \text{RealPair} = \text{Pair}(\text{Real}); \)
  - \( \text{list}(\sigma) = \ldots ; \)
  - \( \text{VAR} \)
  - \( \text{line: list}(\text{Char}); \)

- **Values of a polytype**
- What is the set of values of a polytype? Weird question...

- **In C++**
  - A class template has no values, only if you substitute an actual type to its type variable, you will get a real type.

- **In ML**
  - One can easily define values of a polytypes representing polymorphic functions. For example, the type of the function \( \text{second} \) is the polytype
    - \( \sigma \times \sigma \rightarrow \sigma \).
  - A tough problem—what are the values of the polytype \( \text{list}(\sigma)? \)

- **Definition** The set of values of any polytype is the intersection of all types that can be derived from it.

- **Rationale** Suppose \( v \) is a value of a polytype for which no monotype substitution was performed. Then the only legitimate operations on \( v \) would be those available for any monotype derived from the polytype.
104. Example: polytype list(σ)

Monotypes Derived From list(σ)

- list(Integer) all finite lists of integers, including the empty list.
- list(Boolean) all finite lists of truth values, including the empty list.
- list(String) all finite lists of strings, including the empty list.

The empty list is the only common element

- Nonempty lists are values of a specific monotype, determined by components’ type.
- The empty list is a value of any monotypes derived from list(σ).
- The type of the empty list has type list(σ).
- There are no other values of type list(σ).

105. Example: the polytype σ → σ

Monotypes derived from σ → σ:

- Integer → Integer includes the integer identity function, the successor function, the absolute value function, the squaring function, etc.
- String → String includes the string identity function, the string reverse function, the space trimming function, etc.
- Boolean → Boolean includes the truth value identity function, the logical negation function, etc.

The identity function is common to all σ → σ types. In fact, this is the only such common value.

106. Values of polytypes (more examples)

ϕ(σ) The empty set, ❙

Pointer(σ) The value nil.

σ × σ → σ Function second

(β → γ) × (α → β) → (α → γ) Function o

(σ → σ) → (σ → σ) id, twice, thrice, fourth, etc., and even function fixedpoint (the function mapping any σ → σ function to id : σ → σ).

Pair(σ) = σ × σ empty

Array(σ, σ) = σ → σ empty

107. Polytypes & software engineering

The polytype of a function is very telling of what it does. It is often easy to guess what a function does, just by considering its polytype. Many polytypes have only one value, which eliminates the guessing altogether.

Easy examples

- list(σ) → σ
- list(σ) → list(σ)
- list(σ) → Integer
- σ → σ
- σ × σ → σ
- (β → γ) × (α → β) → (α → γ)

Slightly more difficult

- list(σ) × list(σ) → list(σ × σ),
- (σ → σ) × list(σ) → List(σ),
- (σ × σ → σ) → σ × List(σ) → σ

108. Algebra of polytypes

- There are software systems that promote reuse by supporting a search for functions based on their signatures.
- Clearly, the search must be insensitive to application of the commutative laws to product and choice.
- Further, the search should be made insensitive to choice of labels

4.3.6 Inclusion polymorphism

10 Frames:

- Inclusion polymorphism
- Subtyping
- Inclusion polymorphism
- Subranges in Pascal
- Subtypes in Pascal
- Subtypes in Ada: built-in types
- Subtypes in Ada: array types
- Subtypes in Ada: user-defined types
- Hypothetical ML with structural subtyping
- Non-type parametric polymorphism

109. Inclusion polymorphism

Inclusion Polymorphism: The other kind of universal polymorphism. Arising from an inclusion relation between types or sets of values.

Typing

Polymorphic typing

Monomorphic typing

Universal

Subrange Inheritance

Inclusion

Parameteric

Subtypes

Polytypes

110. Subtyping

Typing

Polymorphic typing

Monomorphic typing

Universal

Subrange Inheritance

Inclusion

Parameteric

Subtypes

Polytypes

Most inclusion polymorphism is due to subtyping, but not always.

Definition (Subtyping: Version I)

Type A is a subtype of the type B if A ⊆ B.

Definition (Subtyping: Version II)

Type A is a subtype of the type B, if every value of A can be coerced into a value of B.

111. Inclusion polymorphism

Builtin:

- Pascal: The Nil value belongs to all pointer types.
- C: The value 0 is polymorphic. It belongs to all pointer types.
• C++: The type `void *` is a super-type of all pointer types.

**User Defined** Two Varieties

(2) Subranges in PASCAL:

- Anything applicable to `Integer` will be applicable to type `Index`.
- Anything applicable to `Char` will be applicable to type `Digit`.

**OO** A subclass is also a subtype

**112. Subranges in Pascal**

```
TYPE
  Index = 1..100;
  Digit = '0'..'9';
```

- Type `MonthLength` has four values: 28, 29, 30, 31.
- Values of make a subset of type `Integer`.
- Any operation that expects an `Integer` value will happily accept a value of type `MonthLength`.
- Type `MonthLength` “inherits” all operations of type `Integer`.

“Inheritance” in Pascal

A PASCAL subrange type “inherits” all the operations of its parent type; otherwise, no PASCAL type inherits any operations from another distinct type.

**113. Subtypes in Pascal**

PASCAL recognizes only one restricted kind of subtype: subranges of discrete atomic types.

```
STATEMENTS
```

**114. Subtypes in Ada: built-in types**

In contrast, ADA allows subtypes of all atomic types, as well as user-defined, composite types.

**Discrete types in Ada**

```
subtype Natural is Integer range 0..Integer’last;
subtype Small is Integer range -3..+3;
```

**Indiscrete types in Ada**

```
subtype Probability is Float range 0.0..1.0;
```

**115. Subtypes in Ada: array types**

```
type String is array (Integer range <> ) of Character;
subtype String5 is String (1..5);
subtype String7 is String (1..7);
```

**116. Subtypes in Ada: user defined types**

```
type Sex is (f, m);
type Person (gender : Sex) is record
  name : String (1..8);
  age : Integer range 0..120;
end record;
subtype Female is Person (gender => f);
subtype Male is Person (gender => m);
```

**117. Hypothetical ML with structural subtyping**

```
type point = {x: real, y: real};
type circle = {x: real, y: real, r: real};
type box = {x: real, y: real, w: real, d: real};
```

Assuming inheritance relationship being derived from structure\(^8\), we have

```
box < circle < point.
```

Operations associated of `point` should be applicable to `box`, e.g.,

```
move: σ ⊆ Point • σ × Real × Real → σ.
```

**118. Non-type parametric polymorphism**

What we have seen so far is...

**Entity** Type (parameterized)

**Parameter** Type

**Output** Type (concrete)

**Entity** Function (parameterized)

**Parameter** Type

**Output** Function (concrete)

\(^8\) In most mainstream PLs, including Java and C++, structure is derived from inheritance relationship

\(^9\) An example was shown above
119. Varieties of polymorphism

Ad Hoc Created by hand; caters for a limited number of types

Overloading A single identifier denotes several functions is an ad hoc term simultaneously
- Reuse is limited to names, but there are is reusable code

Coercion A single function can serve several types thanks to implicit coercion between types
- Extending the utility of a single function, using implicit conversions

Universal Systematic, applies to many types

Parametric Functions that operate uniformly on values of different types

Inclusion Subtypes inherit functions from their supertypes

References

- Ad hoc Polymorphism
- Function Overloading
- Operator Overloading
- Parametric Polymorphism
- Polymorphism
- Type Conversion
- Type Punning

Exercises

1. What (if any) kind of polymorphism?
   (a) operator “>=” in Pascal?
   (b) ... and in C?
   (c) ... and in C++?
   (d) ... and in C?
   (e) ... and in C++?
   (f) “O” in C++?
   (g) “[ ]” in Pascal?

2. Write a Pascal program that while using ranges makes a type error?

3. Is the type error flagged at compile time? At runtime? How?

4. Write a C++ program in which an enum variable is overflowed.

5. Is the type error flagged at compile time? At runtime? How?

6. Repeat the above two questions for Pascal.

7. Pascal has this problem of programs not being able to read files. Explain this problem.

8. With respect to this problem, why does it still make sense to allow functions such as eof which can take many different types of files?

9. Classify the polymorphism kind (if any) of all of Pascal’s predefined functions, constants, and procedures.

4.4 Polymorphism in practice

Contents [34 frames]

4.4.1 Overloading in Pascal, C/C++, and Java [20]
4.4.2 Coercion and the C++ overloading tournament [21]
4.4.3 Polymorphic functions [23]
4.4.4 Type inference [24]
4.4.5 Checking parameters with parametric polymorphism [24]
4.4.6 Case studies [25]

4.4.1 Overloading in Pascal, C/C++, and Java

7 Frames: 
- Keyword overloading in C++
- Built-in operator overloading in C
- Built-in overloading of operator “*” in C
- Built-in operator overloading in Pascal
- User defined operator overloading in C++
- More operator overloading opportunities in C++
- Overloading in Java

Meanings of C++’s “static” keyword are only vaguely related:
### 121. Builtin operator overloading in C

- Keyword overloading does not make entities with more than one type.
- Keyword overloading is not type polymorphism

Many builtin operators offer overloaded semantics

#### Integer multiplication
\( \text{int} \times \text{int} \rightarrow \text{int} \)

#### Long integer multiplication
\( \text{long} \times \text{long} \rightarrow \text{long} \)

#### Floating point multiplication
\( \text{double} \times \text{double} \rightarrow \text{double} \)

#### Pointer dereferencing
\( \text{Pointer}(\sigma) \rightarrow \sigma \) for any type \( \sigma \)

“*” has another overloading in type definitions, but this overloading is not considered polymorphism.

#### 122. Builtin overloading of operator “*” in C

Operator “*” in Pascal serves for

- **Integer negation**: \( \text{Integer} \rightarrow \text{Integer} \)
- **Real negation**: \( \text{Real} \rightarrow \text{Real} \)
- **Integer subtraction**: \( \text{Integer} \times \text{Integer} \rightarrow \text{Integer} \)
- **Real subtraction**: \( \text{Real} \times \text{Real} \rightarrow \text{Real} \)
- **Set difference**: \( \text{Set}(\sigma) \times \text{Set}(\sigma) \rightarrow \text{Set}(\sigma) \), where \( \sigma \) is any of the types for which PASCAL’s \textit{sets} can be created

### 123. Builtin operator overloading in Pascal

Even if you do not know JAVA, you should be able understand and apply the following:

#### Builtin operator overloading: Similar to C++

- “*” serves also for string concatenation.

#### Programmer defined operator overloading: None.

Language designer did not wish to replicate the C++ nightmare.

#### Builtin function overloading: None.

Java just like many other languages has no “builtin” functions.

#### Programmer defined function overloading: Similar to C++.

### 4.4.2 Coercion and the C++ overloading tournament

6 Frames: 
- Coercion in ML
- Programmer defined coercion in C++
- The overloading tournament in C++
- A tournament example
- More tournament examples
- Overloading + coercion + parametric + inclusion = C++ style headache!
Coercion in ML

No mixed type arithmetic in ML:

```ml
- 1+1;
  val it = 2 : int
- 1.0+1.0;
  val it = 2.0 : real
- 1+1.0;
  stdIn:7.1-7.6 Error: operator and operand don't agree [literal]
operator domain: int * int
operand: int * real
in expression: 1 + 1.0
```

No implicit coercion from `int` to `real`; must use function `real`

```ml
- real;
  val it = fn : int -> real
  - (real 1) + 1.0;
    val it = 2.0 : real
```

Programmer defined coercion in C++

Can be done by

- Defining a (non-`explicit`) constructor with a single argument
- Overloading the type cast operator

```cpp
class Rational {
  public:
  Rational(double);
  explicit Rational(const char *s);
  operator double(void);
  ...
};
```

```cpp
Rational r = Rational("half"); // Error
Rational h = Rational("half"); // OK
```

The overloading tournament in C++

In every function call site `foo(a1,a2, ..., an)`, there could be many applicable overloaded versions of `foo`. C++ applies context independent, compile-time tournament to select the most appropriate overload.

Ranking of coercion operations (short version)

None or unavoidable `array → pointer, T → const T, ...
Size promotion `short → int, int → long, float → double, ...
Standard conversion `int → double, double → int, Derived* → Base*
Programmer defined `by constructor or operator overloading
Ellipsis e.g., `int printf(const char *fmt, ...)`

Winner must be:

- Better match in at least one argument
- At least as good for every other argument

An error message if no single winner is found

Resolve ambiguity of the function call

```cpp
max(a,b)
```

where,

- `a` is of type `float`
- `b` is of type `Rational`

and with two candidates:

**I** `double max(double, double)`

**II** `Rational max(long double, Rational)`

Signature `(double, double)`

- 1st argument `float → double`
- 2nd argument `Rational → long double`

Signature `(long double, Rational)`

- 1st argument `float → Rational`
- 2nd argument `none`

First argument equally good (size promotion)

Second argument second contestant wins ("none" is better than "programmer defined")

second contestant wins.

With the declarations made previously, which version of `max` would the following invoke?

```cpp
max(Rational(3),\'\'
```

Given

```cpp
void foo(int) { cout << "int"; }
void foo(char) { cout << "char"; }
void foo(char *) { cout << "char*"; }
void foo(const char *) { cout << "const char*"; }
```

What will be printed?

```cpp
int
```

Overloading + coercion + parametric + inclusion = C++ style headache!

- Parametric polymorphism may contribute to ambiguity

- Inheritance may contribute to ambiguity

- The “overloading” tournament is not limited to overloading

- Certain PLs languages forbid overloading and coercion and restrict parametric polymorphism for precisely this reason.
4.4.3 Polymorphic functions

7 Frames: What are polymorphic functions? write vs. eof in Pascal Polymorphic functions with C++’s templates If Pascal allowed polymorphic functions... Polymorphic functions in ML Polymorphic functions taking function parameters Polymorphic identity function in ML

133. What are polymorphic functions?

Definition 4.4.1 (Polymorphic functions). Functions that can work on a variety of types; a kind of parametric polymorphism i.e., polymorphism occurring for unboundedly many related types. The type variety may or may not show up as an explicit parameter.

134. write vs. eof in Pascal

write(E)

- Effect depends on the type of E: type Char, type String, type Integer...
- The identifier write simultaneously denotes several distinct procedures, each having its own type
- Overloading
  - (We ignore in this course the “magic” of Write taking multiple parameters, where each can be of a different type.)

eof(F)

- Type is: File(σ) → Boolean, where σ is any type
- Function is polymorphic (‘many-shaped’).
- Argument types: File of Char, File of Integer, etc.
- operates uniformly on all of argument types

135. Polymorphic functions with C++’s templates

Definition of a function template

```cpp
template<typename Type>
Type max(Type a, Type b) {
    return a > b ? a : b;
}
```

Using template functions

```cpp
int x,y,z;
double r,s,t;
z = max(x,y);
t = max(r,s);
```

Type Parameters

- Explicitly declared
- Inferred upon use

136. If Pascal allowed polymorphic functions...

function disjoint(s1, s2: set of σ) :Boolean;
begin
    disjoint := (s1 * s2 = [])
end
VAR chars : set of Char;
   ints1, ints2 : set of 0..99;
...
if disjoint(chars, [ ‘a’, ‘e’, ‘i’, ‘o’, ‘u’ ]) then ...
if disjoint(ints1, ints2) then ...

Definition 4.4.2 (Type variables/type parameters). Type expressions like σ in the definition of disjoint are called type variables or type parameters.

137. Polymorphic functions in ML

Type variables are used in ML to define parametric polymorphism: Definition

```ml
fun second(x: τ, y: τ) = y
or
fun second(x,y) = y
```

Type is τ × τ → τ, where τ is arbitrary.

Use

- second(13,true)
- second(name) where name is the pair (1984,"Orwell")

Illegal Use

- second(13)
- second(1983,2,23)

138. Polymorphic functions taking function parameters

Function twice takes as a parameter function f and returns a function g such that g(x)=f(f(x)):

```ml
fun twice(f: σ → τ) = fn : (x: σ) => f( f(x) )
```

e.g.,

```ml
val fourth = twice(sqr)
```
Function \( o \) takes two arguments, functions \( f \) and \( g \) and returns a function which is their composition:

\[
\text{fun } o \circ (f : \beta \rightarrow \gamma, g : \alpha \rightarrow \beta) = \text{fn } (x : \alpha) \Rightarrow f(g(x))
\]

e.g.,

\[
\text{val even = not } o \text{ odd}
\]
or,

\[
\text{fun twice}(f : \sigma \rightarrow \sigma) = f \circ f
\]

139. **Polymorphic identity function in ML**

Identity function \( \sigma \rightarrow \sigma \).

\[
\text{fun id}(x : \sigma) = x
\]

represents

- Identity mapping on booleans
  \[
  \{\text{false} \rightarrow \text{false, true} \rightarrow \text{true}\}
  \]
  \[\text{(4.4.2)}\]

- Identity mapping on integers
  \[
  \{0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 2, \ldots\}
  \]
  \[\text{(4.4.3)}\]

- Identity mapping on strings
  \[
  \{\varepsilon \rightarrow \varepsilon, \text{"a"} \rightarrow \text{"a", } \text{"b"} \rightarrow \text{"b", } \ldots, \text{"aa"} \rightarrow \text{"aa", } \text{"ab"} \rightarrow \text{"ab", } \ldots\}
  \]
  \[\text{(4.4.4)}\]

4.4.4 Type inference

140. **Type inference**

The type of an entity is inferred, rather than explicitly stated.

Pascal

Constant definition:

\[
\text{CONST } \pi = 3.14159264590;
\]

1. \( 3.14159264590 \) is of type \textit{Real}.
2. Therefore, \( \pi \) is of type \textit{Real}.

ML

Function definition

\[
\text{fun even}(n) = (n \mod 2 = 0)
\]

1. \( \mod \) is of type \( \text{int} \times \text{int} \rightarrow \text{int} \).
2. Since \( n \) occurs in \( n \mod 2 \), \( n \) is of type \textit{int}.
3. The type of operator \( = \) is \( \sigma \times \sigma \rightarrow \text{bool} \) for all \( \sigma \);
4. \( n \) occurs in \( n \mod 2 \), so \( n \) is of type \textit{int}.
5. Therefore, the type of \( n \mod 2 = 0 \) is \textit{bool}
6. It follows that the type of \( \text{even} \) is

\[
\text{int} \rightarrow \text{bool}
\]

Define a \texttt{max} function in ML:

\[
\text{val max = fn : int \times int \rightarrow int}
\]

But we want \texttt{max} to operate on reals:

\[
\text{val max = fn : real \times real \rightarrow real}
\]

142. **Polymorphic type inference**

Type inference might yield a monotype

- As for the function \( \text{even} \)

Type inference might yield a polytype

- \[
  \text{fun } id(x) = x
  \]
  - The type of \( \text{id} \) is \( \sigma \rightarrow \sigma \)

- \[
  \text{fun } o \circ (f, g) = \text{fn } (x) \Rightarrow f(g(x))
  \]
  - We can see from the way they are used that \( f \) and \( g \) are functions.
  - The result of \( g \) must be the same as the argument type of \( f \).
  - Thus, type of \( o \) can be inferred:

\[
\sigma \rightarrow \gamma \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)
\]

4.4.5 Checking parameters with parametric polymorphism

6 Frames:

\[
\square \text{ Checking type parameters: in C++} \quad \square \text{ Checking type parameters in ML} \quad \square \text{ Polymorphic functions: C++ vs. ML} \quad \square \text{ Parametric polymorphism: ML vs. Ada vs. C++} \quad \square \text{ Const exercises} \quad \square \text{ Polytypes in Ada: generics}
\]

143. **Checking type parameters: in C++**

Templates are checked when they are instantiated, not when they are defined:

```
// An, a third template function
struct S {…} s1, s2, s3 = max(s1,s2);
```

```
gcc max.C
max.C: In instantiation of
 'const T& max(const T &a, const T &b) { return a > b ? a : b; }'
max.C:7:25: required from here
max.C:3:14: error: no match for ‘operator’>
 operand types are ‘const ’T and ‘const ’T’
 return a > b ? a : b;
```

```
template <typename T> // a ’function’ ’template'
const T& max(const T &a, const T &b) { return a > b ? a : b; } // and, a third template function
```
Polymorphic functions are checked when they are defined, not when they are used.

Cannot define a polymorphic max function, since most types do not have a “greater than” operator, and the language does not offer overloading.

### Polytypes in Ada: generics

```ada
generic(type ElementType) module Stack;
export Push, Pop, Empty, StackType, MaxStackSize;
constant MaxStackSize = 10;
type private StackType =
    record
        Size: 0..MaxStackSize := 0;
        Data: array 1..MaxStackSize of ElementType;
    end;
procedure Push(
    reference ThisStack: StackType;
    readonly What: ElementType);
procedure Pop(reference ThisStack): ElementType;
procedure Empty(readonly ThisStack): Boolean;
end;
−− Stack
module IntegerStack = Stack(integer);
```

### 4.4.6 Case studies

5 Frames:
- Case study: universal pointer in C
- Case study: casting in C++
- Const exercises
- Parametric polymorphism on enumerated types in Pascal
- Responses to inflexibility in Java

### 149. Case study: universal pointer in C

**Universal pointer type.** In C, a `void` pointer could be assigned to any pointer, and any pointer can be assigned to `void*`.

```c
extern void* malloc(size_t);
extern void free(void*);
void foo(size_t n) {
    long *buff = malloc(n * sizeof(long));
    free(buff);
}
```

**Parametric Polymorphism** In C the coercion from `long*` to `void*` and vice-versa is not ad-hoc

- It universally exists for all pointer types
- The actions performed are the same for all pointer types
150. Case study: casting in C++

C++ deprecates C-style casts; instead there are four cast operations:

- **const_cast<σ>** takes a type σ and returns a cast operator from any type σ to σ provided only that σ can be obtained from σ just by adding const.

- **reinterpret_cast<σ>** takes a type σ and returns a cast operator from any type σ to σ (useful for peeping into bit representations).

- **static_cast<σ>** takes a type σ and returns a cast operator from any type σ, provided this is a standard casting (e.g. double to int).

- **dynamic_cast<σ>** takes a type σ of a derived class and returns a cast operator from any type σ of its base classes into σ.

151. Const exercises

- Given are the following definitions:
  
  ```
  typedef char* t1;
  typedef char* const t2;
  typedef const char* t3;
  typedef const char* const t4;
  t1 c1;
  t2 c2;
  t3 c3;
  t4 c4;
  ```

- Determine for all i, j, k which of the following commands will legally compile?
  
  - c_i = c_j;
  - c_i = const_cast<t_j>(c_k);
  - *c_i = *c_j;
  - *const_cast<t_j>(c_j) = *c_k;

152. Parametric polymorphism on enumerated types in Pascal

Nonsense code to demonstrate Pascal’s built-in parametric polymorphism:

```pascal
for m := January to December do
  for d := Saturday downto Sunday do
    case suit of
      Club, Heart:
        suit := succ(suit);
      Diamond, Spade:
        if suit < Heart then
          if ord(m) < ord(d) then
            suit := pred(suit);
        end;
```
List of equations

4.3.1 Motivation

(4.3.1) In a monomorphic type systems, functions (and other entities) have precisely one type. .......... 11

\[ f \text{ is a function } \Rightarrow |\text{types}(f)| = 1. \]

(4.3.2) Function gcd in Pascal is monomorphic ... 11

\[ |\text{types}(\text{gcd})| = |\{\text{Integer} \times \text{Integer} \rightarrow \text{Integer}\}| = 1 \]

(4.3.3) Function gcd in C is monomorphic .......... 12

\[ |\text{types}(\text{gcd})| = |\{\text{int} \times \text{int} \rightarrow \text{int}\}| = 1 \]

4.4.1 Overloading in Pascal, C/C++, and Java

(4.4.1) One of the overloaded meanings of operator “-” follows parametric polymorphism ............... 21

\[ \forall \sigma \in \text{T}_{\text{Pascal}} \bullet \left( (\rho \times \sigma \rightarrow \rho \sigma) \rightarrow \rho \sigma \right) \in \text{types}(\text{“-”}) \]

4.4.3 Polymorphic functions

(4.4.2) Identity mapping on booleans ............... 24

\[ \{\text{false} \rightarrow \text{false}, \text{true} \rightarrow \text{true}\} \]

(4.4.3) Identity mapping on integers ............... 24

\[ \{\ldots, -2 \rightarrow -2, -1 \rightarrow -1, 0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 2, \ldots\} \]

(4.4.4) Identity mapping on strings ............... 24

\[ \{ \varepsilon \rightarrow \varepsilon, \quad \text{“a”} \rightarrow \text{“a”}, \text{“b”} \rightarrow \text{“b”}, \ldots, \quad \text{“aa”} \rightarrow \text{“aa”}, \text{“ab”} \rightarrow \text{“ab”}, \ldots, \quad \ldots \} \]

4.4.4 Type inference

(4.4.5) Type of operator ok in ML ............... 24

\[ o \in (\beta \rightarrow \gamma) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma) \]