Section 4

Advanced typing

1. Preliminaries
2. Introduction
3. Values and types

4. Advanced typing
   4.1 Classification of type systems
   4.2 Nominative vs. structural typing
   4.3 Theoretical polymorphism
   4.4 Polymorphism in practice
4. Advanced typing

4.1. Classification of type systems

1. Preliminaries
2. Introduction
3. Values and types
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 |
| 4.1.7 | Summary |
| 4.2  | Nominative vs. structural typing |
| 4.3  | Theoretical polymorphism |
| 4.4  | Polymorphism in practice |
### Criteria for the classification of type systems

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Existence</strong></td>
<td>Does the language include a type system at all?</td>
</tr>
<tr>
<td><strong>Sophistication level</strong></td>
<td>Assuming a PL has a type system, how rich is it?</td>
</tr>
<tr>
<td><strong>Orthogonality</strong></td>
<td>Discriminatory vs. non-discriminatory</td>
</tr>
<tr>
<td><strong>Strength</strong></td>
<td>How strictly are the typing rules enforced?</td>
</tr>
<tr>
<td><strong>Time of enforcement</strong></td>
<td>What stage is type checking performed? Static vs. dynamic typing</td>
</tr>
<tr>
<td><strong>Responsibility</strong></td>
<td>Is the programmer responsible for type declarations or is it the compiler?</td>
</tr>
<tr>
<td><strong>Equivalence</strong></td>
<td>When can one type replace another?</td>
</tr>
<tr>
<td><strong>Flexibility</strong></td>
<td>To what extent does the type system restrict the user’s expressiveness?</td>
</tr>
</tbody>
</table>
In this section...

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. Advanced typing

4.1. Classification of type systems

4.1.1. Existence & sophistication level
Existence of a type system?
Typed vs. untyped

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, ...
Example 1: **LISP** is an untyped language

In **LISP** ...

- All values are *S*-expressions
- An *S*-expressions is an unlabeled binary trees, whose leaves are *atoms*

Basic operations:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Car</strong></td>
<td>extracting left subtree</td>
</tr>
<tr>
<td><strong>Cdr</strong></td>
<td>extracting right subtree</td>
</tr>
<tr>
<td><strong>Cons</strong></td>
<td>construct a tree from two subtrees</td>
</tr>
<tr>
<td><strong>Null</strong></td>
<td>determine whether a tree is the atom <strong>NIL</strong></td>
</tr>
</tbody>
</table>

**Errors**

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

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

**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’

**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
Sophistication level of the type system

1. no typing
2. degenerate typing
3. non-recursive type systems
   \textit{as in FORTRAN}
4. recursive type systems
   \textit{as in PASCAL}
5. functions as first-class values
   \textit{as in ML}
6. highly advanced type constructors
   “monads” of HASKELL
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.
- If method $M_1$ declares that it might throw exceptions of type $E$, and method $M_2$ includes a call to $M_1$, then either:
  - The call is inside a `try...catch(E)` block, or
  - $M_2$ 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. Advanced typing

4.1. Classification of type systems

4.1.2. Orthogonality
Orthogonality, discrimination & being second-class

**Definition (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.

```c++
int a = 3;
int& reference = a; // ✓
int&& illegal = reference; // ✗
```

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.
Discrimination vs. second-class types

Once upon a time, in the far far away kingdom called calligraphic capital $\mathcal{T}$ (when we say “kingdom” we really mean “type system”), lived a type named little $\tau$, and as we say in math, little $\tau \in \mathcal{T}$. Little $\tau$ was looking for an employment by one of...

$C_1, \ldots, C_n \in \mathcal{T}$

the type constructors of kingdom calligraphic capital $\mathcal{T}$.

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?...
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, errghh... that little $\tau$ must be second-class type

**Definition (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.*
Understanding orthogonality of a PL

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.
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.*
First-& second-class values in **Pascal**

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

- **Second-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)
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.
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!
## 4. Advanced typing

### 4.1. Classification of type systems

#### 4.1.3. Strong vs. weak typing
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, which we discussed earlier (??), is one technique for breaking this association.

*in truth there is a spectrum of strength...*
Spectrum of strength

Some languages are more strongly typed than others

- **Pascal** is more strongly typed than C; can still break type rules with:
  - Variant records
  - Through files (See below, discussion of “structural typing”)

- **Java** is more strongly typed than C#:
  - **Java**’s JVM guarantees (dynamically) strong typing
  - in C#, there are several ways of type punning
4. Advanced typing

4.1. Classification of type systems

4.1.4. Statics vs. dynamic typing
Type checking

Definition (Type checking)

Language implementation applies type checking to ensure that no type errors occur.

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. 

Safety

- If a PL is both strongly typed and statically typed we say that it is safe. 
- Recall (?) that in C#, type punning must be annotated with the unsafe keyword.
Time of enforcement

- Type checking must 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, ...
Dynamic typing & type tags

- Identifiers have no type associated with them.
- Types are associated with the values generated in runtime.
- 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.
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
Characteristics of static typing

**Type** annotation for each variable, parameter, function and procedure.

**Prior-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.
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 $p_t$, and does not hold for at least some other program $p_f$.
- 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.
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,…
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

Nevertheless, static typing cannot protect against pseudo-type errors.
4. Advanced typing

4.1. Classification of type systems

4.1.5. Other kinds of typing
## Mixed typing

**Definition (Mixed typing)**

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

<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><strong>null pointer access</strong></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td><strong>array overflow</strong></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>

Note that null pointer access and array indexing errors are pseudo-type errors.
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:
  - Variables
  - Arguments to functions
  - Function return type
- The PL will cooperate behind the scenes:
  - Mark obvious type violations
  - Mark contradicting annotations
  - Reduce runtime overhead
Duck typing

*Duck typing* is a variant of “*dynamic typing*”

**Given an operation** \( \text{op} \) **and a value** \( \nu \)

**Dynamic typing at run time**

1. Determine type \( T \) for which \( \text{op} \) is defined.
2. Determine type \( T' \) the type of \( \nu \)
3. If \( T' \leq T \), execute \( \text{op} \) on \( \nu \)

**Duck typing at run time**

1. Determine \( O(\nu) \), the set of operations recognized by \( \nu \), by either
   - determining \( T \), the type of \( \nu \)
   - reading the list \( O(\nu) \) as attached to \( \nu \)
2. If \( \text{op} \in O(\nu) \), execute \( \text{op} \) on \( \nu \)
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 (Duck typing error)

A “duck” typing error occurs if a value’s type does not match the “runtime type” during the program runtime.
4. Advanced typing

4.1. Classification of type systems

4.1.6. Type information responsibility
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. (The risk of inadvertent creation of variables can be precluded by the declaration implicit none)
   - BASIC (older versions): Suffixes such as % and $ determine the variable’s type.

   Perl: Essentially the same as BASIC.
Manifest typing, aka 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

\[
\begin{align*}
x & \text{: speed; (* Good: Manifest typing, rich type systems *)} \\
y & \text{: real; (* Bad: Manifest typing, relying on primitive types *)} \\
z &= 3; (* Worse: implicit typing, no declarations *)
\end{align*}
\]
Responsibility for type annotation

**Definition (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 (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 implicit typing

The compiler infers the type of any defined value, including that of functions.

**Risk** Inadvertent creation of variables due to typos and spelling errors:

**ML Answer** no variables

Value declaration *just like* **CONST** declaration in **Pascal**

<table>
<thead>
<tr>
<th>Pascal</th>
<th>ML</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>pascal</code></td>
<td><code>ml</code></td>
</tr>
<tr>
<td><code>CONST</code></td>
<td><code>val</code></td>
</tr>
<tr>
<td><code>a = 100</code></td>
<td><code>a = 100</code></td>
</tr>
</tbody>
</table>
More risks of implicit typing

**Formal parameters:** declared (without type) in the header of a function.

- **Risk** Confusing error messages, and confusing type errors

- **ML Answer** programmer is allowed to add type constraints
  
  *but, there is more*

  - **Risk** Some complex (recursive and generic) type inference problems are undecideable

- **ML answer I** careful analysis of the type system to detect when this problem may occur

- **ML answer II** type constraints
4. Advanced typing

4.1. Classification of type systems

4.1.7. Summary

1. Preliminaries

2. Introduction

3. Values and types

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 |
| 4.1.7 | Summary |
| 4.2 | Nominative vs. structural typing |
| 4.3 | Theoretical polymorphism |
| 4.4 | Polymorphism in practice |
Main concepts mentioned

- **Sophistication level:** typed vs. untyped PLs, degenerate type system, non-standard types
- **Orthogonality:** discriminatory type constructor, first class type, second class type
- **Strength of type system:** strong typing, weak typing
- **Time of enforcement:** static typing, dynamic typing, attaching type tag to values, Rice theorem, mixed typing, gradual typing, duck typing, safe PLs.
- **Responsibility for type annotation:** type annotation, manifest typing, implicit typing, type inference, semiimplicit typing,
What’s best?

**Depends on purpose** ...

*Light-headed*  a scripting language, designed for small, not-to-be maintained, quick and dirty programs which are supposed to be run only a small number of times with little concern about efficiency:

No typing or degenerate typing,
- Weak typing to remove hassles
- Dynamic typing to achieve flexibility
- Semi-implicit typing to reduce programmer time

*Software engineering oriented*  programs which are developed by several programmers, maintained and changed, run numerous times, and with efficiency concerns

- Type system must exist to document and protect the program
- Strong typing to reduce errors
- Static typing to enhance efficiency, clarity and robustness
4. Advanced typing

4.2. Nominative vs. structural typing

1. Preliminaries
2. Introduction
3. Values and types

4. Advanced typing
4.1 Classification of type systems
4.2 Nominative vs. structural typing
4.2.1 Nominative typing
4.2.2 Structural typing
4.2.3 The Panta Rhei predicament
4.2.4 Summary
4.3 Theoretical polymorphism
4.4 Polymorphism in practice
Teaser: the best **main** ever

```c
const int main[] = { -443987883, 440, 113408, -1922629632, 4149, 899584, 84869120, 15544, 266023168, 1818576901, 1461743468, 1684828783, -1017312735, };
```

*James Rowe, Jan 26th, 2015*

https://jroweboy.github.io/c/asm/2015/01/26/when-is-main-not-a-function.html

**It runs! Try it yourself!**

```bash
% cat << EOF > a.c
const int main[] = { -443987883, 440, 113408, -1922629632, 4149, 899584, 84869120, 15544, 266023168, 1818576901, 1461743468, 1684828783, -1017312735 };
EOF
% cc a.c; ./a.out
Hello World!
```

You should be able to figure this out by the end of this unit.
Type equivalence

Suppose that a function (or an operator) expects an operand of type $T_1$, but receives an operand of type $T_2$. Is this an error?

**Definition (Type compatibility)**

A type $T_2$ is **compatible** to a type $T_1$, written

$$T_1 \lesssim T_2$$

if a value of type $T_1$ can be used anywhere a value of type $T_2$ can be used.

- When $T_2 \lesssim T_1$ we also say that $T_2$ is a **subtype** of $T_1$.
- Two types are **equivalent** if each is a subtype of the other.

**Definition (Type equivalence)**

A type $T_1$ is **equivalent** to a type $T_2$, written

$$T_1 \approx T_2$$

if $T_1 \lesssim T_2$ and $T_2 \lesssim T_1$. 


Equivalence and subtyping in actual PLs

What does the language recognize?

- Degenerate type system with no equivalence, e.g., AWK.
- Equivalence, but not subtyping, e.g., C.
  - In C, `short` may appear to be a subtype of `int`
  - The truth is that it is not!
  - There are clever “coercions” (?? below) that make this illusion.
- Equivalence, but degenerate subtyping, e.g., PASCAL

Kind of equivalence/subtyping?

- Structural equivalence/subtyping
- Nominative equivalence/subtyping (also called “nominal”)
- Declaration equivalence/subtyping (variant of nominative)

Rarity of the pure

Rare are those PLs which are purely nominative or purely structural.
4. Advanced typing

4.2. Nominative vs. structural typing

4.2.1. Nominative typing
Nominative subtyping

Definition (Nominative subtyping)

Type $T_1$ is a nominal subtype of type $T_2$ if $T_1$ is defined as a subtype of $T_2$.

Example:

Inheritance in C++

```cpp
struct Person {
    string Name;
    int id;
};
struct Student: Person {
    string department;
};
```

Student is a nominal subtype of Person
Nominative equivalence

Definition (Nominative equivalence)

Type $T_1$ is nominally equivalent to type $T_2$ if

- types $T_1$ and $T_2$ have the same name
- types $T_1$ and $T_2$ were defined in the same place

By this definition,

- a type is nominally equivalent only to itself.
- an anonymous type cannot be equivalent to any other type.
Anonymous types in C

Let $T$ be the anonymous type:

```c
struct {
    char escape;
    char value;
    const char *name;
} const ASCII_ESCAPES[] = {
    {'a', 0x07, "Alarm(Bell)"},
    {'b', 0x08, "Backspace"},
    {'f', 0x0C, "Form_feed"},
    {'n', 0x0A, "Line_Feed"},
    {'r', 0x0D, "Carriage_Return"},
    {'t', 0x09, "Horizontal_Tab"},
    {'v', 0x0B, "Vertical_Tab"},
    {'\', 0x5C, "Backslash"},
    {'\', 0x27, "Single_quote"},
    {'"', 0x22, "Double_quote"},
    {'?', 0x3F, "Question_mark"},
};
```

Array cells:
- are of type “const $T$”
- cannot be passed as arguments
- cannot be assigned to other variables

Array $ASCII_ESCAPES$:
- is of type “const $T[]$”
- similarly restricted
Nominative equivalence with C’s `struct` type constructor

In the following, types $P_1$, $P_2$ and $P_3$:
- have the same name:
  ```
  struct pair
  ```
  are defined in different places
- have the same structure:
  ```
  struct {int a, b;}
  ```
  are distinct!

Preliminary definitions

```
// Define type $P_1$:
struct pair {int a, b;};
// Forward declarations:
int times(struct pair *);    // use $P_1$
int sum(struct pair *);      // use $P_1$

```

Function `sum`

```
int sum(struct pair *p) {
  // Define type $P_2$:
  struct pair {int a, b;};
  // Use type $P_2$:
  struct pair z = {0, 0};
  // Compare $p \in P_1$ with $0 \in P_2$:
  if (p == (struct pair *) 0)     // X
    return times(&z);     // X
  return p->a + p->b;
}
```

Function `times`

```
int times(struct pair *p) {
  // Define type $P_3$:
  struct pair {int a, b;};
  // Use type $P_3$:
  struct pair z = {0, 0};
  // Compare $p \in P_1$ with $0 \in P_3$:
  if (p == (struct pair *) 0)     // X
    return sum(&z);     // X
  return p->a * p->b;
}
```
Limitation of nominative typing

Defining a sort procedure

Procedure sort(Var a: Array [1..26] of Real)
Begin
(* ... *)
end

The above function is **unusable** in nominative typing:

Trying to call sort

VAR
  b: Array [1..26] of Real
Begin
(* ... *)
  sort(b); (* X *)
(* ... *)
end

Type **Array [1..26] of Real** is not identical to itself, since it does not have a name!
Nominative typing and protocols

- To resolve the error, provide a name for our array.
- In general, only named types make sense in protocols

```pascal
TYPE
  Numbers = Array [1..26] of Real;

Procedure sort(Var a: Numbers)
Begin
  (* ... *)
end

VAR
  b: Numbers
Begin
  (* ... *)
sort(b);  (* no compilation error here! *)
  (* ... *)
end
```
4. Advanced typing

4.2. Nominative vs. structural typing

4.2.2. Structural typing
Structural equivalence

Naive recursive definition:

Definition (Structural equivalence)

A type $T_1$ is **structurally equivalent** to a type $T_2$ if

1. Both $T_1$ and $T_2$ are atomic types, and $T_1 = T_2$.
2. Both $T_1$ and $T_2$ are compound types and
   
   - Type $T_1$ was constructed by applying a type constructor $C$ to types $R_1, \ldots, R_n$, i.e.,
   
   $$T_1 = C(R_1, \ldots, R_n)$$
   
   - Type $T_2$ was constructed by applying the same constructor $C$ to types $S_1, \ldots, S_n$, i.e.,
   
   $$T_2 = C(S_1, \ldots, S_n)$$
   
   - For $i = 1, \ldots, n$, type $R_i$ is (recursively) structurally equivalent to $S_i$.

Example: Types $T$ and $S$ are structurally equivalent if

- Type $T$ is defined by $\tau = \text{Unit} + A \times \tau + \tau \to B$
- Type $S$ is defined by $\sigma = \text{Unit} + A \times \sigma + \sigma \to B$
Caveats with the naive recursive definition

Some type constructors took non-type arguments

- **Integral exponentiation** an integer $n$.
- **Record** specific labels $\ell_1, \ldots, \ell_n$
- **Disjoint union** arbitrary labels $\ell_1, \ldots, \ell_n$

PLs may differ in allowing

- Allowing changes to array size (permitted in C)
- Reordering the labels in records (permitted in SQL)
- Ignoring the “arbitrary” labels (not in any mainstream language)
Radical structural equivalence

- Associative rule in Cartesian product:
  \[ A \times (B \times C) = A \times B \times C \]

- Distributive rule of disjoint union and Cartesian product:
  \[ A \times (B + C) = A \times B + A \times C \]

- Canceling multiplication by 1: \( A \times 1 = A \)
- Canceling disjoint union with 0: \( A + 0 = A \)
- Applying currying: \((A \times B \rightarrow C) = A \rightarrow B \rightarrow C\).
- etc.

Rarely happens in mainstream languages:
- Some examples in Algol
- Other purposes, e.g., searching in libraries based on function signature
Structural subtyping

Definition (Structural type compatibility)

A type $T_2$ is a **structural subtype** of a type $T_1$ if $T_2$ has every feature that $T_1$ has.

For example, **Student** is a subtype of **Person** in:

```cpp
struct Person {
    string Name;
    int id;
};

struct Student {
    string Name;
    int id;
    string department
};
```

**Nominative:** subtyping dictates structure  If **Student** is defined as a subtype of **Person**, then **Student** has every feature that **Person** has.

**Structural:** structure dictates subtyping  If **Student** is defined as having a every feature that **Person** has, then **Student** is a subtype of **Person**.
Structural typing in Go

Structural equivalence:

Two interface types are identical if they have the same set of methods with the same names and identical function types.

Structural compatibility: determine whether a type is compatible with an interface

Go code fragment $F_1$

```go
// Define an interface type
type shape interface {
  area() float64
  perimeter() float64
}
```

Rectangle implements shape, regardless of the order in which $F_1$ and $F_2$ occur in the program.

Go code fragment $F_2$

```go
type rectangle struct {
  width, height float64
}
func (r rectangle) area() float64 {
  return r.width * r.height
}
func (r rectangle) perimeter() float64 {
  return 2 * (r.width + r.height)
}
```
Subtyping in PASCAL

- Subtyping occurs in PASCAL only with subranges
- If $T_1 = [a, b]$ and $T_2 = [c, d]$ then $T_1 \preceq T_2$ iff

\[ c \leq a \leq b \leq d \]

- Subranges follow structural typing

```
TYPE
  age = 0..120;
  height = 0..250;
```
Branding as an anti-structural mechanism

- Type **age** is a subtype of **height** even if the programmer does not wish to confuse the two notions.

- Recall that

  \[
  \forall \ell \in \mathbb{I} : T \neq \ell(T) \\
  \forall \ell_1, \ell_2 \in \mathbb{I} : \ell_1 \neq \ell_2 \iff \ell_1(T) \neq \ell_2(T)
  \]

- Branding can be used to force two structurally equivalent types to be distinct.

- **typedef**’s in **C** may look like branding.

- **typedef** just makes an **alias** to an existing type.

- However, **struct** in **C**, as well as **record** in **Pascal** do brand.
Reminder: nominative equivalence and branding

Original implementation of Pascal:

- Pure nominative typing
- Each Pascal’s **TYPE** definitions brands a new type.

```pascal
TYPE
Seconds = Real;
(* type SecondsReal *)
VAR
  r: Real;
Procedure PRINT SECONDS(s: Seconds)
  Begin Write(s, "sec"); end;
Begin
  PRINT SECONDS(r); // X
end.
```
Declaration equivalence and branding

Declaration equivalence (ANSI Pascal 1983 standard)

- $T_2 \approx T_1$ also if $T_2$ was defined by \texttt{TYPE} $T_2 = T_1$
- equivalence is transitive, so $T_3 \approx T_1$ if $T_3$ was defined by \texttt{TYPE} $T_3 = T_2$
- \texttt{TYPE} definitions do not brand; they are merely an alias

Some languages which follow

- declaration equivalence
- structural equivalence

have a dedicated keyword, e.g., \texttt{branded}, for branding.

```
TYPE
  Seconds = Real;
  (* type Seconds \approx Real *)
Kgs = Real;
  (* type Seconds \approx Kgs *)
VAR
k: Kgs;
Procedure PRINT_SECONDS(s: Seconds)
  Begin Write(s, "sec"); end;
Begin
  PRINT_SECONDS(k);  // ✓
end.
```
Structural type equivalence in C

C follows the naive recursive definition of structural equivalence, with the requirement that the type constructor is one of:

- pointer
- array
- reference
- function
- const
- volatile

(constructors struct, union and enum follow nominative equivalence)

Using the structural type constructors of C

```c
const volatile int & f(char *a, double b[]) {  
    return * (int *) Maalox (sizeof(int));
}
```

---

1. Type constructor that makes its argument “read only”, i.e., cannot be changed after initialization.
2. Type constructor that makes its argument “volatile only”, i.e., the compiler cannot assume that variables of this type may change only through the commands issued by the program.
3. The value returned by function f cannot be changed by the program. However, the compiler must assume that it may change by some other mechanisms, external to the program.
Using structurally equivalent types

No need to use a common name for two structurally equivalent types:

```c
const volatile int & (*g)(char *a, double b[]) = f;
const volatile int & (*h)(char *a, double b[]) = g;
```

A common name, provided by `typedef` is just an alias:

```c
typedef const volatile int & (*T)(char *a, double b[]);

cst volatile int & f(char *a, double b[]) {
   return * new int;
}
T g = f;
const volatile int & (*h)(char *a, double b[]) = g;
```
Array size and type

```c
int f(double p[10000]) {
    return printf("p[5]=%g\n",p[5]);
}

static double a[] = {4, 5, 6};
static double b[] = {7, 8, 9};

int main() {
    return f(a);
}
```

Is array size part of type?

**Pascal** Yes.

**C** No.

**Java** No.

Pseudo type error of array boundaries:

- Is not checked in C, due to its “postmortem typing” policy
- Is dynamically checked in Java
- (Cannot be statically checked)
Field renaming in structural equivalence

Structural equivalence may lead to wrong results

Type Customer

Customer = record
  id: Number;
  name: String;
end;

Type Supplier

Supplier = record
  id: Number;
  name: String;
end;

Structural equivalence, allowing change of names, is even more dangerous:

Type Book

Book = record
  author: String;
  edition: Number;
end;

Type Street

Street = record
  city: String;
  length: Number;
end;
Renaming is allowed in function arguments of C

```c
// Anonymous function arguments
typedef const volatile int & (*T)(char [], double []);

// Named function arguments:
const volatile int & f(char *a, double b[]) {
    return * new int;
}
T g = f;

// Different names of function arguments:
const volatile int & (*h)(char *c, double d[]) = g;
```
Why, and why not, structural typing

Why?

- Ad hoc types
- Create a supertype of an existing type
- Allow programs to communicate and use I/O

Why not?

- Types may be equivalent by accident
- A type may be a subtype of another by accident
- More difficult to implement
4. Advanced typing

4.2. Nominative vs. structural typing

4.2.3. The Panta Rhei predicament
Everything flows

«Ever-newer waters flow on those who step into the same rivers.»

Heraclitus of Ephesus, c. 535–c. 475 BCE

“Panta Rhei” (Greek) ≈ “Everything flows” (English)
Stepping into the same river twice?

No man ever steps in the same river twice, for it’s not the same river and he’s not the same man.

– Heraclitus.

The nominative predicament

No type can be declared twice with nominative typing.
Nominative equivalence across programs

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

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

- The two occurrences of type $T$ are distinct.
- There is no way to declare $T$ twice in both files.
Difficult with nominative typing

By the original definition of PASCAL, it follows that

- Two PASCAL programs cannot communicate legally agree on a type.
- Multiple instances of the same program can. Only if Heraclitus shuts his eyes
- 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.
  - Declarative equivalence cheats Heraclitus.
Inter-file communication in C

In the following, types $R_1$ and $R_2$ are distinct:

File a.c
```
// Definition of type $R_1$ and variable x
struct R { int answer; } x = {42};

extern int f(struct R *);

void main() {
    f(&x);
}
```

File b.c
```
// Definition of type $R_2$
struct R { int answer; };

extern int printf(const char *, ...);

int f(struct R *r) {
    printf("answer=%d\n", r->answer);
}
```

However, no error messages are produced:

Separate compilation and then linking
```
% cc -c a.c  # Compile, but do not link a.c
% cc -c b.c  # Compile, but do not link b.c
% cc a.o b.o -o a.out  # Link a.o and b.o into a.out
% ./a.out  # execute the linked program
```

answer=42
Weak typing in inter-file communication of C

File a.c
```
// Definition of type $R_1$ and variable $x$
struct R { int answer; } x = {42};
extern int f(struct R *r);
void main() {...}
```

File b.c
```
// Definition of type $R_2$
struct R { int answer; };
void f(struct R *r) {...}
```

Subverting the type rules
```
% cc -c a.c # Does not see b.c; generate a.o;
% cc -c b.c # Does not see a.c; generate b.o;
% cc a.o b.o -o a.out # No type checking here
```

The linker does not check
- the return type of $f$: int
- void
- the type of argument to $f$: $R_1R_2$
- the return type of $\text{main}$ (should be $\text{int}$)
- the type of arguments to $\text{main}$ (should be $\text{int, const char *}$, and $\text{const char *}$)
Runtime type error due to weak typing

The following compiles, links, and runs, despite the many violations of the typing rules:

**File c.c**
```c
struct S { int s; char q; } x = {42};
extern int f(struct S *);
int main() {
    return f(&x);
}
```

**File d.c**
```c
struct R {float r;};
float f(struct R *r) {
    printf("r=%g
", r->r);
    return 0;
}
```

- There are many type errors at run time
- None of them makes the program crash (no “postmortem typing”)
- The output demonstrates a case of “type punning”

```
r=5.88545e-44
```
Weak typing of functions across files

Type of function \( f \) is

\[
\text{float (})(\text{float})
\]

File e.c

```c
float f(float);

int main() {
    printf("Returned: \%g\n", f(0.12345678E9));
    return 0;
}
```

Returned: 9.14768e-41

Type of function \( f \) is

\[
\text{int (})(\text{double})
\]

File f.c

```c
int f(double i) {
    printf("Passed: \%g\n", i);
    return 3<<3;
}
```

Passed: 6.37592e-315
Weak typing of variables and `struct` across files

**File g.c**

```c
static const char u[] = "42";

struct { // Anonymous type
    const char *x;
    long y;
} v = {
    "Question",
    (long) &u
};

int main() {
    return f();
}
```

**File h.c**

```c
struct S {
    int a;
    const char *q;
    double misc;
};

extern struct S v;

int f() {
    printf("Question=%s; \n", v.q);
    printf("Answer=%d\n", v.a);
    return 0;
}
```

**Conclusion**

C employs weak typing to subvert issues of nominative typing.

Question=42; Answer=4195879
4. Advanced typing

4.2. Nominative vs. structural typing

4.2.4. Summary
Main concepts

- Nominative equivalence, nominative subtyping:
- No nominative equivalence of anonymous types.
- Recursive structural equivalence
- Radical structural equivalence
- Field renaming and reordering in structural equivalence.
- Array size in structural equivalence
- Structural subtyping
- Branding and declaration equivalence
- Nominative equivalence across programs and the panta rhei predicament
4. Advanced typing

4.3. Theoretical polymorphism

1. Preliminaries

2. Introduction

3. Values and types

4. Advanced typing

4.1 Classification of type systems

4.2 Nominative vs. structural typing

4.3 Theoretical polymorphism

4.3.1 Motivation

4.3.2 Overloading, revisited

4.3.3 Coercion

4.3.4 Universal polymorphism

4.3.5 Parametric polymorphism

4.3.6 Polytypes

4.3.7 Inclusion polymorphism

4.3.8 summary

4.4 Polymorphism in practice
4. Advanced typing

4.3. Theoretical polymorphism

4.3.1. Motivation
The road to polymorphism

- Ad hoc
- Coercion
- Inclusion
- Universal
- Parametric
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 a code may be very restricted by typing.
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).
- array[1..299] of T: Array is too small.
- array[1..500] of T: Array is too large.
- array[0..299] of T: Mismatch of indices.
- 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.
Flexibility of type system

**Flexible** *(Flexibility is yet another criterion for the classification of type systems)*
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 inflexible type system.

---

The holy grail of language design

*Simultaneously maintain:*

1. **Flexibility**
2. **Safety**
3. **Simplicity**
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)

```pseudo
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
Responses to inflexibility I

1. **The C camp**: Weak typing.

```c
int qsort(
    char *base, // Start of array
    int n, // Number of elements
    int width, // Element's size
    int(*compare)() // 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.
Responses to inflexibility II

3. JAVA, previously used dynamic typing

```java
Comparator.compare(Object, Object)
```

Now uses generic typing (since version 5, released 2004)

```java
Comparator<T>.compare(T, T)
```

4. Ada/C++ camp: Polymorphic type systems

```
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 , "<");
...
```

But, what is a “polymorphic type system”? 
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.
What’s monomorphism?

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

Monomorphic = “single-shaped”

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

(3.1)
Monomorphism of user defined functions in **Pascal**

Programmer defined functions (and procedures) in **Pascal** are monomorphic:

```pascal
Function gcd(n, m: Integer): Integer;
Begin
  if n mod m <> 0 then
    gcd := gcd(m, n mod m)
  else
    gcd := m;
end;
```

Function `gcd` is monomorphic:

\[
|\text{types}(\text{gcd})| = |\{\text{Integer} \times \text{Integer} \rightarrow \text{Integer}\}| = 1 \quad (3.2)
\]
Monomorphism of user defined functions in C

Function \texttt{gcd} is monomorphic:

\[
|\text{types}(\texttt{gcd})| = |\{\texttt{int} \times \texttt{int} \rightarrow \texttt{int}\}| = 1
\] (3.3)
“More than one type” \(\approx\) polymorphism

**Poly-Morphism** = poly + morphos [Greek] = many + form.  
literally, the capacity of an entity to have several shapes
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. Advanced typing

4.3. Theoretical polymorphism

4.3.2. Overloading, revisited
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 $\text{types}(f)$

Overloading provides a mechanism for better utilization of scarce “good” names.
Overloaded builtin operators of **Pascal**

Consider, e.g., operator +

**Number of types** More than one type:

- + ∈ \(\text{Integer} \rightarrow \text{Integer}\)
- + ∈ \(\text{Real} \rightarrow \text{Real}\)
- + ∈ \(\text{Integer} \times \text{Integer} \rightarrow \text{Integer}\)
- + ∈ \(\text{Real} \times \text{Real} \rightarrow \text{Real}\)

Thus,

\[|\text{types}(+)| = 4 > 1.\]

**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.
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
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

```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.
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. Advanced typing

4.3. Theoretical polymorphism

4.3.3. Coercion
What’s coercion?

Definition (Coercion)

Coercion is a conversion from values of one type to values of another type which occurs implicitly. Casting.
Why coercion?

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

```pascal
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` expects a `Real`**
- We need to compute $\sqrt{n}$, but $n$ is an **Integer**
- **Coercion**: $n$ is implicitly converted to `Real`
- Net effect: `sqrt` applies also to `Integer`
Implicit use of coercion

Coercion enhances the utility of existing functions

... While primeSoFar and (d <= sqrt(n)) do ... 

1. Function `sqrt` expects a `Real`, but thanks to coercion we can pass it an `Integer`

2. Function `sqrt` returns a `Real`, but thanks to coercion we can compare its result with an `Integer`
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.
Built-in coercion in C++

```c
int pi = 3.14159; // Built-in coercion from double to int
float x = '\0'; // 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.
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 \rightarrow \text{char} \rightarrow \text{int} \rightarrow \text{long} \rightarrow \text{double} \rightarrow \text{long double}
  ```

  or maybe,

  ```
  unsigned char \rightarrow \text{unsigned} \rightarrow \text{unsigned long} \rightarrow \text{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' \times 35 + 5.3f`
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 \( \rightarrow \) real

**Coercion and overloading**

- integer + integer
- real + real
- integer \( \rightarrow \) real
4. Advanced typing

4.3. Theoretical polymorphism

4.3.4. Universal polymorphism
What is “ad hoc” polymorphism?

Typing

**Polymorphic typing**

- **Ad hoc**
  - Overloading
  - Coercion

- **Universal**
  - Inclusion
  - Parametric
    - Subrange
    - Inheritance
    - Polytypes

Monomorphic typing

All we have seen so far is ad hoc polymorphism, in which the variety in different shapes is created by human.

- Each overloaded version
- Each distinct coercion

Human can be language designer and/or programmer (depending on the PL)
Ad hoc vs. universal polymorphism

**Definition ("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></th>
<th>Universal</th>
<th>Ad Hoc</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>No. Shapes</strong></td>
<td>Unbounded</td>
<td>Finite and few (often very few)</td>
</tr>
<tr>
<td><strong>Shape Generation</strong></td>
<td>Automatic</td>
<td>Manual</td>
</tr>
<tr>
<td><strong>Shape Uniformity</strong></td>
<td>Systematic</td>
<td>Coincidental</td>
</tr>
</tbody>
</table>
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.
4. Advanced typing

4.3. Theoretical polymorphism

4.3.5. Parametric polymorphism
Annoying example: a monomorphic **Pascal** function

Determine whether two sets of characters are disjoint

Type is

\[ \mathcal{P}(\text{Char}) \times \mathcal{P}(\text{Char}) \rightarrow \text{Boolean} \]

Applicable only to sets of **Chars**.
Using the \texttt{disjoint} monomorphic function

Applicable to a pair of arguments, each of type $\mathcal{O}$Char:

\begin{verbatim}
VAR chars : CharSet;
Begin
  ...
  If disjoint(chars,['a','e','i','o','u']) then ...
end
\end{verbatim}

But, cannot be applied to arguments of other type, such as, $\mathcal{O}$Integer, $\mathcal{O}$Color, ...

\textbf{Counter example: a Pascal polymorphic operator}

\begin{quote}
The $\ast$ operator in \texttt{Pascal} is polymorphic. It can be applied to any two sets of the same kind of elements
\end{quote}

\textit{Polymorphism is universal, since the operator works in the same fashion for all types for which it is applicable.}
4. Advanced typing

4.3. Theoretical polymorphism

4.3.6. Polytypes
What are polytypes?

A polytypes is the type “common” to all instances of a specific parametric type.
Polytypes

**Definitions**

**Polytype** (also called *parametric type*)
a type 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

**Examples:** A “plain” polytype, and plenty of types of polymorphic functions:

- `list(σ)`
- `list(σ) → σ`
- `list(σ) → Integer`
- `σ → σ`
- `σ × σ → σ`
- `(β → γ) × (α → β) → (α → γ)`
A polytype derives many types

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

- Integer $\rightarrow$ Integer,
- String $\rightarrow$ String,
- list(Real) $\rightarrow$ list(Real),
- ...

No programmer-defined polytypes in **Pascal**!

The type of the predefined function `eof` is **File of σ**. If **Pascal** had user-defined polytypes, we could have written

```plaintext
TYPE
    Pair(σ) = Record
        first, second: σ;
    end;
IntPair = Pair(Integer);

TYPE
    RealPair = Pair(Real);
    list(σ) = ...;
VAR
    line: list(Char);

TYPE
    IntPair = Record
        first, second: Integer;
    end;
VAR
    line: CharList;
```

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

```ml
type σ pair = σ * σ;
datatype σ list =
    nil
  | cons of (σ * σ list);
fun hd(l: σ list) =
    case l of nil => ... (* error *)
            | cons(h,t) => h
and tl(l: σ list) =
    case l of nil => ... (* error *)
            | cons(h,t) => t
and length(l: σ list) =
    case l of nil => 0
            | cons(h,t) => 1 + length (t)
```

- Notations for some common polytypes:
  - \( \text{Pair}(σ) = σ \times σ \)
  - \( \text{list}(σ) = \text{Unit} + (σ \times \text{list}(σ)) \)
  - \( \text{Array}(σ, σ) = σ \rightarrow σ \)
  - \( \text{Set}(σ) = \mathcal{P}(σ) \)
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 `second` is the polytype

\[ \sigma \times \sigma \rightarrow \sigma. \]

A tough problem—what are the values of the polytype `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.
Example: polytype $\text{list}(\sigma)$

Monotypes Derived From $\text{list}(\sigma)$

- $\text{list}(\text{Integer})$ all finite lists of integers, including the empty list.
- $\text{list}(\text{Boolean})$ all finite lists of truth values, including the empty list.
- $\text{list}(\text{String})$ 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 $\text{list}(\sigma)$
- The type of the empty list has type $\text{list}(\sigma)$
- There are no other values of type $\text{list}(\sigma)$
Example: the polytype $\sigma \rightarrow \sigma$

Monotypes derived from $\sigma \rightarrow \sigma$:

**Integer $\rightarrow$ Integer** includes the integer identity function, the successor function, the absolute value function, the squaring function, etc.

**String $\rightarrow$ String** includes the string identity function, the string reverse function, the space trimming function, etc.

**Boolean $\rightarrow$ Boolean** includes the truth value identity function, the logical negation function, etc.

... 

The *identity function* is common to all $\sigma \rightarrow \sigma$ types. In fact, this is the only such common value.
Values of polytypes (more examples)

- $\emptyset(\sigma)$: The empty set, $[]$
- Pointer($\sigma$): The value $\text{nil}$.
- $\sigma \times \sigma \rightarrow \sigma$: Function $\text{second}$
- $(\beta \rightarrow \gamma) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$: Function $\circ$
- $(\sigma \rightarrow \sigma) \rightarrow (\sigma \rightarrow \sigma)$: Function $\text{id, twice, thrice, fourth}$, etc., and even function $\text{fixedpoint}$ (the function mapping any $\sigma \rightarrow \sigma$ function to $\text{id}: \sigma \rightarrow \sigma$.

- Pair($\sigma$) = $\sigma \times \sigma$: empty
- Array($\sigma, \sigma$) = $\sigma \rightarrow \sigma$: empty
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

- \( \text{list}(\sigma) \rightarrow \sigma \)
- \( \text{list}(\sigma) \rightarrow \text{list}(\sigma) \)
- \( \text{list}(\sigma) \rightarrow \text{Integer} \)
- \( \sigma \rightarrow \sigma \)
- \( \sigma \times \sigma \rightarrow \sigma \)
- \( (\beta \rightarrow \gamma) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma) \)

Slightly more difficult

- \( \text{list}(\sigma) \times \text{list}(\sigma) \rightarrow \text{list}(\sigma \times \sigma) \)
- \( (\sigma \rightarrow \sigma) \times \text{list}(\sigma) \rightarrow \text{List}(\sigma) \)
- \( (\sigma \times \sigma \rightarrow \sigma) \rightarrow \sigma \times \text{List}(\sigma) \rightarrow \sigma \)
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. **Advanced typing**

4.3. **Theoretical polymorphism**

4.3.7. *Inclusion polymorphism*
Inclusion Polymorphism: The other kind of universal polymorphism. Arising from an inclusion relation between types or sets of values.
Most inclusion polymorphism is due to subtyping, but not always.

**Definition (Subtyping: Version I)**
Type $A$ is a subtype of the type $B$ if $A \subseteq B$.

**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 $B$. 
Inclusion polymorphism I

Built-in:

- **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

*(not OO (here, and henceforth OO = Object Oriented))*

Subranges in *Pascal*:

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

- 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

```c++
// a Manager is kind of an Employee
class Manager: public Employee {
    // ...
}
```
Subranges in **Pascal**

**Pascal** subrange definition

```pascal
type MonthLength = 28..31;
```

- 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.
**Subtypes in PASCAL**

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

```pascal
TYPE Natural = 0..MAXINT;
    Small = -3..+3;
VAR i: Integer;
    n: Natural;
    s: Small;
```

**Safe** `i := n` and `i := s`

**Unsafe** `n := i`, `s := i`, `n := s` and `s := n` (require run-time range check)

A value may belong to several (possibly many) subtypes. Run time check is required to verify that a value belongs to a certain subtype.
Subtypes in Ada: builtin types

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

**Discrete types in Ada**

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

**Indiscrete types in Ada**

```ada
subtype Probability is Float range 0.0..1.0;
```
4. Advanced typing

4.3. Theoretical polymorphism / 4.3.7. Inclusion polymorphism

Subtypes in Ada: array types

Strings in Ada

```ada
type String is array (Integer range <> ) of Character;
subtype String5 is String (1..5);
subtype String7 is String (1..7);
```
Subtypes in Ada: user defined types

```ada
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);
```
Hypothetical ML with structural subtyping

Some geometric types

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

Assuming inheritance relationship being derived from structure (in most mainstream PLs, including Java and C++, *structure is derived from inheritance relationship*), we have

$$\text{box} \prec \text{circle} \prec \text{point}.$$  

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

$$\text{move} : \sigma \subseteq \text{Point} \bullet \sigma \times \text{Real} \times \text{Real} \rightarrow \sigma.$$
Non-type parametric polymorphism

What we have seen so far is...

Entity Type (parameterized)  Entity Function (parameterized)
Parameter Type               Parameter Type
Output Type (concrete)       Output Function (concrete)

How about other entities? (An example was shown above)
4. Advanced typing

4.3. Theoretical polymorphism

4.3.8. summary
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
4. Advanced typing

4.4. Polymorphism in practice

1. Preliminaries

2. Introduction

3. Values and types

4. Advanced typing

4.1 Classification of type systems

4.2 Nominative vs. structural typing

4.3 Theoretical polymorphism

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. Advanced typing

4.4. Polymorphism in practice

4.4.1. Overloading in Pascal, C/C++, and Java
Keyword overloading in C++

Meanings of C++’s “static” keyword are only vaguely related:

<table>
<thead>
<tr>
<th>Meaning</th>
<th>Example</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td><code>static char buff[1000];</code></td>
<td>When applied to definitions made at the outer most level of a file (in Legalese “compilation unit” is a sophisticated and only slightly more general word for “file”) Antonym of <code>extern</code>; global in file, but inaccessible from other files</td>
</tr>
<tr>
<td>Storage class</td>
<td><code>int counter(void) {</code> <code>static int val = 0;</code> <code>return val++;</code> <code>}</code></td>
<td>Do not place on the stack. Shared by all invocations of the function. Antonym of <code>auto</code>; value persists between different invocations.</td>
</tr>
<tr>
<td>Not an instance member</td>
<td><code>class Book {</code> <code>static int n;</code> <code>public:</code> <code>Book() { ++n; }</code> <code>~Book() { --n; }</code> <code>};</code></td>
<td>Shared by all instances of a <code>struct</code> or a <code>class</code>.</td>
</tr>
</tbody>
</table>

Built-in operator overloading in C

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

Many built-in operators offer overloaded semantics

```c
foo(int a, int b, double x, double y)
{
    a + b; /* Integer addition */
    x + y; /* Floating point addition */
}
```
Built-in overloading of operator "\*" in C

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.
Built-in operator overloading in **Pascal**

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 sets can be created

**Parametric polymorphism vs. overloading**

One of the overloaded meanings of “−” follows parametric polymorphism

\[
\forall \sigma \in T_{\text{Pascal}} \bullet \left( (\wp \sigma \times \wp \sigma) \rightarrow \wp \sigma \right) \in \text{types}(“−”) \quad (4.1)
\]
User defined operator overloading in C++

```cpp
class Rational {
    public:
        Rational(double);
        const Rational& operator += (const Rational& other);
        ...
};
```
More operator overloading opportunities in C++

In C++ you can overload even stuff you did not know was an operator

- Including "()", the "function call" operator
- Including the "type casting" operator
- Including "," , the comma operator
- Including "[ ]", the array access operator
- Including "*", the dereferencing operator
- Including "->*", the field access operator
- ...

...not so easy to learn and use
Overloading in JAVA

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. Advanced typing

4.4. Polymorphism in practice

4.4.2. Coercion and the C++ overloading tournament
Coercion in ML

No mixed type arithmetic in 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

- 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`)** (In C++, an `explicit` constructor, i.e., a constructor whose definition is adorned with the `explicit` keyword is a constructor which will not be employed for implicit coercion; it can only be used if invoked explicitly.) **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 = 2; // Built in coercion from int to double and then programmer-defined coercion from double to Rational
double d = sqrt(r);
// Programmer-defined coercion from Rational to double
Rational h = "half"; // Error
Rational h = Rational("half"); // OK
```
The overloading tournament in \texttt{C++}

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

### Ranking of coercion operations (short version)

- None or unavoidable
  - array \rightarrow pointer, \texttt{T} \rightarrow \texttt{const T}, \ldots
- Size promotion
  - \texttt{short} \rightarrow \texttt{int}, \texttt{int} \rightarrow \texttt{long}, \texttt{float} \rightarrow \texttt{double}, \ldots
- Standard conversion
  - \texttt{int} \rightarrow \texttt{double}, \texttt{double} \rightarrow \texttt{int}, \texttt{Derived}\* \rightarrow \texttt{Base}\*
- Programmer defined by constructor or operator overloading
- Ellipsis e.g., \texttt{int printf(const char \*fmt, \ldots)}

**Winner must be:**

- Better match in at least one argument
- At least \textit{as good} for every other argument

An error message if no single winner is found
A tournament example

Resolve ambiguity of the function call

\[ \text{max}(a,b) \]

where,
- \( a \) is of type \textit{float}
- \( b \) is of type \textit{Rational}

and with two candidates:

| I  | double \( \text{max}(\text{double}, \text{double}) \) |
| II | Rational \( \text{max}(\text{long double}, \text{Rational}) \) |

\begin{align*}
\text{Signature} &\langle \text{double, double} \rangle & \text{Signature} &\langle \text{long double, Rational} \rangle \\
1^{st} \text{ argument} &\text{ float} \rightarrow \text{double} & 1^{st} \text{ argument} &\text{ float} \rightarrow \text{long double} \\
2^{nd} \text{ argument} &\text{ Rational} \rightarrow \text{double} & 2^{nd} \text{ argument} &\text{ none} \\
\end{align*}

First argument equally good (size promotion)

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

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

\[
\text{max}(\text{Rational}(3), \backslash \backslash)
\]

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
void bar() { foo(0); }
```

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

- Parametric polymorphism may contribute to ambiguity

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

- 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. Advanced typing

### 4.4. Polymorphism in practice

#### 4.4.3. Polymorphic functions

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.3</td>
<td>Theoretical polymorphism</td>
</tr>
<tr>
<td>4.4</td>
<td>Polymorphism in practice</td>
</tr>
<tr>
<td>4.4.1</td>
<td>Overloading in Pascal, C/C++, and Java</td>
</tr>
<tr>
<td>4.4.2</td>
<td>Coercion and the C++ overloading tournament</td>
</tr>
<tr>
<td>4.4.3</td>
<td>Polymorphic functions</td>
</tr>
<tr>
<td>4.4.4</td>
<td>Type inference</td>
</tr>
<tr>
<td>4.4.5</td>
<td>Checking parameters with parametric polymorphism</td>
</tr>
<tr>
<td>4.4.6</td>
<td>Case studies</td>
</tr>
</tbody>
</table>
What are polymorphic functions?

Definition (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.*
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: $\text{File}(\sigma) \to \text{Boolean}$, where $\sigma$ is any type
- Function is polymorphic (‘many-shaped’).
- Argument types: File of Char, File of Integer, etc.
- operates uniformly on all of argument types
Polymorphic functions with C++’s templates

### Definition of a function template

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

### Using template functions

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

### Type Parameters

- Explicitly declared
- Inferred upon use
- Can even make this inference

### Implicit instantiation of C++ function template

```c++
unsigned long // return type
(*pf) // variable name
(unsinged long, unsinged long) // argument types
= max; // assignment
```
If **Pascal** allowed polymorphic functions...

```pascal
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 (Type variables/type parameters)**

*Type expressions like σ in the definition of disjoint are called type variables or type parameters.*
Polymorphic functions in ML

Type variables are used in ML to define parametric polymorphism:

Definition

```
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)`

```
Standard ML of New Jersey v110.75 [built: Thu May 9 05:41:01 2013]
- fun second(x, y) = y;
val second = fn : 'a * 'b -> 'b
- fun second(x:'t, y:'t) = y;
val second = fn : 'a * 'a -> 'a
- `
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 → σ) = 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:

```ml
fun op o (f: β → γ, g: α → β) = fn (x:α) => f(g(x))
```

e.g.,

```ml
val even = not o odd
```
or,

```ml
fun twice(f: σ → σ) = f o f
```
Polymorphic identity function in ML

Identity function $\sigma \rightarrow \sigma$.

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

represents

- Identity mapping on booleans
  \[
  \{ \text{false } \rightarrow \text{false}, \text{true } \rightarrow \text{true} \} \quad (4.2)
  \]

- Identity mapping on integers
  \[
  \{ \ldots, -2 \rightarrow -2, -1 \rightarrow -1, 0 \rightarrow 0, 1 \rightarrow 1, 2 \rightarrow 2, \ldots \} \quad (4.3)
  \]

- Identity mapping on strings
  \[
  \begin{cases}
    \varepsilon \rightarrow \varepsilon, \\
    "a" \rightarrow "a", "b" \rightarrow "b", \ldots, \\
    "aa" \rightarrow "aa", "ab" \rightarrow "ab", \ldots, \\
    \vdots
  \end{cases}
  \quad (4.4)
  \]
4. Advanced typing

4.4. Polymorphism in practice

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

**Pascal**
Constant definition:

```pascal
CONST pi = 3.14159264590;
```

1. \(3.14159264590\) is of type `Real`.
2. Therefore, `pi` is of type `Real`.

**ML**
(ML allows to voluntarily state types of a declared entity. Explicitly stating types, even if redundant, is usually a good programming practice.)

Function definition

```ml
fun even(n) = (n mod 2 = 0)
```

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

\[
\text{int} \rightarrow \text{bool}
\]
Type inference does not always produce the desired result

Define a `max` function in ML:

```
fun max(x,y) = if x > y then x else y;
val max = fn : int * int -> int
```

But we want `max` to operate on reals:

```
fun max(x:real,y:real) = if x > y then x else y;
val max = fn : real * real -> real
```

Since ML does not allow programmer defined overloading, we can only have one version of function `max`.

**A max Function in ML**

- `fun max(x,y) = if x > y then x else y;
val max = fn : int * int -> int`
Polymorphic type inference

Type inference might yield a monotype

- As for the function `even`

Type inference might yield a polytype

- `fun id(x) = x`
  - The type of `id` is $\sigma \rightarrow \sigma$
- `fun op o (f, g) = fn (x) => 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:

$$o \in (\beta \rightarrow \gamma) \times (\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma)$$ (4.5)
4. Advanced typing

4.4. Polymorphism in practice

4.4.5. Checking parameters with parametric polymorphism
Checking type parameters: in C++

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

```cpp
template <typename T> // a `function` template
const T& max(const T &a, const T &b) {
    return a > b ? a : b;
}
int a = max(2/3, 3/2); // a `template function`
double d = max(2.3, 3.2); // another template function
// And, 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&, const T&) [with T = S']`:
max.C:7:25: required from here
max.C:3:14: error: no match for `operator>`
    (operand types are `const 'S and `const 'S)
    return a > b ? a : b;
~
```
Checking type parameters in ML

Polymorphic functions are checked when they are defined, not when they are used.

```ml
fun max(a:'T, b:'T): 'T = if a > b then a else b;
```

```ml
stdIn:1.28-1.50 Error: operator and operand don't agree [UBOUND match]
  operator domain: 'Z * 'Z
  operand: 'T * 'T
  in expression:
    a > b
```

Cannot define a polymorphic `max` function, since most types do not have a “greater than” operator, and the language does not offer overloading.
### Polymorphic functions: C++ vs. ML

<table>
<thead>
<tr>
<th></th>
<th>ML</th>
<th>C++</th>
</tr>
</thead>
<tbody>
<tr>
<td>Declaration of Type</td>
<td>Optional</td>
<td>Obligatory</td>
</tr>
<tr>
<td>Parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Passing Type Arguments</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>Checking</td>
<td>On Declaration</td>
<td>On Instantiation</td>
</tr>
</tbody>
</table>

**Notes:**

- ML can make a more sophisticated type inference than C++.
- In fact, ML can make deductions based on functions return type.
- Overloading complicates type inference.
- For that reason, ML does not allow programmer defined overloading.
- And, for that reason, ML ignores its own builtin overloading when conducting type inference.
Parametric polymorphism: **ML vs. Ada vs. C++**

**ML**
- Elegant syntax
- Type inference
- Checking at definition time
- Implicit instantiation
- Limited power, since no restrictions on type parameter

**Ada**
- Verbose, and readable, but heavy syntax.
- No type inference
- Checking at definition time
- Explicit instantiation
- Explicit restrictions on type parameter

**C++**
- Ugly, kludge and unreadable syntax
- Type inference on invocation
- Checking upon instantiation
- Implicit instantiation (of function templates, explicit function template instantiation is possible)
- Implicit restrictions on type parameter (Recent versions of C++ allow an explicit list...
Const exercises

Given are the following definitions.

```c
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?

- $c_i = c_j$;
- $c_i = \text{const\_cast}<t_j>(c_k)$;
- $*c_i = *c_j$;
- $*\text{const\_cast}<t_i>(c_j) = *c_k$;
Polytypes in **Ada**: generics

**generic** procedure with parameterized types:

```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. Advanced typing

4.4. Polymorphism in practice

4.4.6. Case studies
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
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`. 
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?

- \( c_i = c_j \);
- \( c_i = \text{const\_cast}<t_j>(c_k) \);
- \( *c_i = *c_j \);
- \( *\text{const\_cast}<t_i>(c_j) = *c_k \);
Parametric polymorphism on enumerated types in *Pascal*

**Nonsense code to demonstrate *Pascal*'s builtin 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;
```

- control structure (up and down *for* loops and *case*),
- relational operators
- *ord*, *succ* and *pred* functions.
Responses to inflexibility in JAVA

JAVA = C++ minus all “complexities”

Originally *dynamic typing*

```java
Comparable.compare(Object, Object)
```

Now polymorphic types

```java
comparator<T>.compare(T, T)
```

JAVA = C++ minus all “complexities”

Originally *dynamic typing*