

Types

Akim Demaille Étienne Renault Roland Levillain
first.last@lrde.epita.fr

EPITA — École Pour l'Informatique et les Techniques Avancées

March 13, 2017

- 1 What is Type Checking?
- 2 Why using Types?
- 3 The Concepts of Types
- 4 (Crash) Introduction to Natural Deduction

What is Type Checking?

1 What is Type Checking?

2 Why using Types?

3 The Concepts of Types

4 (Crash) Introduction to Natural Deduction

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
 - Booleans
 - User-defined enumerations
 - Subranges
 - Arrays (static, stack dynamic, heap dynamic)
 - Unions (discrimated/free)
 - Structures/Records/Objects
 - Tuples/Lists
 - References/Pointers
 - etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

The data types of a language are a large part of what determines that language's style and usefulness (along with control structures).

- Numerics
- Booleans
- User-defined enumerations
- Subranges
- Arrays (static, stack dynamic, heap dynamic)
- Unions (discrimated/free)
- Structures/Records/Objects
- Tuples/Lists
- References/Pointers
- etc.

Type Checking is the activity of ensuring that the operands of an **operator** are of compatible types

A **compatible type** is one that is

- either legal for the operator
- or allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type

Type Checking is the activity of ensuring that the operands of an **operator** are of compatible types

A **compatible type** is one that is

- either legal for the operator
- **or** allowed under language rules to be implicitly converted by compiler-generated code (or the interpreter) to a legal type

Coercion is the automatic (implicit) conversion from a type to another.

There are 2 kind of coercion in C-like languages:

- **widening** conversions: from a "smaller" type to a "larger one"

```
int i = 42;  
float f = i;
```

- **narrowing** conversions: from a "larger" type to a "smaller one"

```
float f = 42.0;  
int i = f;
```

Note: Java only allows assignment type widening coertions.

Strong Typing

A programming language is **strongly typed** if type errors are always detected.

- the types of all operands can be determined, either at **compile time** or at **runtime**
- detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type
- Ada is nearly strongly typed due to *Unchecked_Conversion*
- C and C++ are not strongly typed languages because both include union types
- F# and ML are strongly typed

Strong Typing

A programming language is **strongly typed** if type errors are always detected.

- the types of all operands can be determined, either at **compile time** or at **runtime**
- detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type
- Ada is nearly strongly typed due to *Unchecked_Conversion*
- C and C++ are not strongly typed languages because both include union types
- F# and ML are strongly typed

Strong Typing

A programming language is **strongly typed** if type errors are always detected.

- the types of all operands can be determined, either at **compile time** or at **runtime**
- detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type
- Ada is nearly strongly typed due to *Unchecked_Conversion*
- C and C++ are not strongly typed languages because both include union types
- F# and ML are strongly typed

Strong Typing

A programming language is **strongly typed** if type errors are always detected.

- the types of all operands can be determined, either at **compile time** or at **runtime**
- detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type
- Ada is nearly strongly typed due to *Unchecked_Conversion*
- C and C++ are not strongly typed languages because both include union types
- F# and ML are strongly typed

Strong Typing

A programming language is **strongly typed** if type errors are always detected.

- the types of all operands can be determined, either at **compile time** or at **runtime**
- detection, at run time, of uses of the incorrect type values in variables that can store values of more than one type
- Ada is nearly strongly typed due to *Unchecked_Conversion*
- C and C++ are not strongly typed languages because both include union types
- F# and ML are strongly typed

Type Equivalence

Two types are **equivalent** if an operand of one type in an expression is substituted for one of the other type, without coercion.

In other words, **Type equivalence** is a strict form of compatibility type compatibility without coercion.

- **Name type equivalence**

- two variables have equivalent types if they are defined either in the same declaration or in declarations that use the same type name

```
int i = 42; int j = 51;
```

- **Structure type equivalence**

- two variables have equivalent types if their types have identical structures

```
using celcius = int;  
int i = 42; celcius j = 51;
```

Why using Types?

1 What is Type Checking?

2 Why using Types?

3 The Concepts of Types

4 (Crash) Introduction to Natural Deduction

Why Using Types

- Types are not necessary:
 - There is none at machine/assembly level operators are “typed” though
 - There are type-less languages e.g., in Tcl or M4 everything is a string
- But they are useful:
 - More control from the compiler
 - Catching “impossible but expressible” situations
 - Optimizing
 - Abstraction (arrays, records, etc.)
 - Memory management (automatic or not)

Why Using Types

- Types are not necessary:
 - There is none at machine/assembly level operators are “typed” though
 - There are type-less languages e.g., in Tcl or M4 everything is a string
- But they are useful:
 - More control from the compiler
 - Catching “impossible but expressible” situations
 - Optimizing
 - Abstraction (arrays, records, etc.)
 - Memory management (automatic or not)

Of FORTRAN and Satellites

ANTENNA.F

```
...
...
DO 1 I = 1, 5

C Extend the antenna.
...
1  If antenna is extended
2  Then Go to 3
...
3  CONTINUE
4  ...
```

antenna.c

```
int I;
...
for (I = 1; I <= 5; ++I)
{
    /* Extend the antenna. */
    ...
    if (antenna_is_extended)
        goto 3
    ...
3:
    ...
}
```

Of FORTRAN and Satellites

ANTENNA.F

```
...
...
DO 1 I = 1, 5

C Extend the antenna.
...
1  If antenna is extended
2  Then Go to 3
...
3  CONTINUE
4  ...
```

antenna.c

```
int I;
...
for (I = 1; I <= 5; ++I)
{
    /* Extend the antenna. */
    ...
    if (antenna_is_extended)
        goto 3
    ...
3:
    ...
}
```

Of FORTRAN and Satellites

ANTENNA.F

```
...
...
DO 1 I = 1. 5

C Extend the antenna.
...
1 If antenna is extended
2 Then Go to 3
...
3 CONTINUE
4 ...
```

antenna.c

```
float D01I = 1.5;

/* Extend the antenna. */
...
if (antenna_is_extended)
    goto 3
...

3: ...
```

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Russel's Paradox

$$E = \{x \notin x\} \quad E \in E \quad E \notin E$$

Based on the conjunction of:

- Any predicate is an object
- Any predicate can be applied to any object

Rejecting one leads to:

- Type theory (1909)
- Zermelo Fraenkel's set theory (1922)

Reject Impossible Values

- 1 1
- $\omega = \lambda x.xx$

Enable Optimizations

- Static types enables static bindings.
- E.g., + in C requires no runtime checks.

The Concepts of Types

1 What is Type Checking?

2 Why using Types?

3 The Concepts of Types

- Numeric Types
- Other Primitive Types
- Complex Types

4 (Crash) Introduction to Natural Deduction

Decreasing safety, increasing flexibility:

- during compilation (*static binding*)
- during loading
- when entering a subprogram
- when executing an instruction

Atomic, builtin

- Logical (Boolean)
- Numerical (integer, float, fixed, complex etc.)
- Character

User defined

- intervals
- enumerations
- arrays
- structures, records
- unions, variants

Atomic, builtin

- Logical (Boolean)
- Numerical (integer, float, fixed, complex etc.)
- Character

User defined

- intervals
- enumerations
- arrays
- structures, records
- unions, variants

1 What is Type Checking?

2 Why using Types?

3 The Concepts of Types

- **Numeric Types**
- Other Primitive Types
- Complex Types

4 (Crash) Introduction to Natural Deduction

Floats

- 1, sign
- 7, exponent
- 24, mantissa

Floats ANSI/IEEE Std 754-1985

[ANSI/IEEE, 1987, Goldberg, 1991]

IEEE Standard for Binary Floating Point Arithmetic

	FLT_	DBL_
RADIX	2	
MANT_DIG	24	53
DIG	6	15
MIN_EXP	-125	-1021
MIN_10_EXP	-37	-307
MAX_EXP	128	1024
MAX_10_EXP	+38	308
MIN	1.17549435E-38F	2.2250738585072014E-308
MAX	3.40282347E+38F	1.7976931348623157E+308
EPSILON	1.19209290E-07F	2.2204460492503131E-016

Floats are surprising (dangerous?)

foo.c

```
#include <stdio.h>

#define TEST(Op) \
    if ((num / den) Op quot) \
        puts("1 / 3 " #Op " 1 / 3");

int main()
{
    float quot = 1.0 / 3.0;
    volatile float num = 1, den = 3;

    TEST(<); TEST(<=);
    TEST(>); TEST(>=);
    TEST(==); TEST(!=);
}
```

Floats are surprising (dangerous?)

foo.c

```
#include <stdio.h>

#define TEST(Op) \
    if ((num / den) Op quot) \
        puts("1 / 3 " #Op " 1 / 3");

int main()
{
    float quot = 1.0 / 3.0;
    volatile float num = 1, den = 3;

    TEST(<); TEST(<=);
    TEST(>); TEST(>=);
    TEST(==); TEST(!=);
}
```

Optimized

```
% gcc-3.4 foo.c \
    -O1 -o foo-c
% ./foo-c
1 / 3 < 1 / 3
1 / 3 <= 1 / 3
1 / 3 != 1 / 3
```

Floats are surprising (dangerous?)

foo.c

```
#include <stdio.h>

#define TEST(Op) \
    if ((num / den) Op quot) \
        puts("1 / 3 " #Op " 1 / 3");

int main()
{
    float quot = 1.0 / 3.0;
    volatile float num = 1, den = 3;

    TEST(<); TEST(<=);
    TEST(>); TEST(>=);
    TEST(==); TEST(!=);
}
```

Optimized

```
% gcc-3.4 foo.c \
    -O1 -o foo-c
% ./foo-c
1 / 3 < 1 / 3
1 / 3 <= 1 / 3
1 / 3 != 1 / 3
```

Not Optimized

```
% gcc-3.4 foo.c \
    -o foo-c
% ./foo-c
1 / 3 > 1 / 3
1 / 3 >= 1 / 3
1 / 3 != 1 / 3
```


Floats are surprising (dangerous?)

From GCC's documentation

On 68000 and x86 systems, for instance, you can get paradoxical results if you test the precise values of floating point numbers. **For example, you can find that a floating point value which is not a NaN is not equal to itself.**

This results from the fact that the floating point registers hold a few more bits of precision than fit in a 'double' in memory. Compiled code moves values between memory and floating point registers at its convenience, and moving them into memory truncates them.

You can partially avoid this problem by using the `-ffloat-store` option.

- PL/I.

```
DECLARE X FIXED DECIMAL (6, 2);  
ON_CHECK(X): Y = X + 12;
```

- Essential in COBOL: “decimals are money”.

- Ada user defined precision.

```
type money is delta 0.01 range 0.0 .. 9999.99;
```

Were builtin at the beginning (FORTRAN).

Other Primitive Types

1 What is Type Checking?

2 Why using Types?

3 The Concepts of Types

- Numeric Types
- **Other Primitive Types**
- Complex Types

4 (Crash) Introduction to Natural Deduction

- Numeric
- Boolean (since ALGOL 60)
- Characters
- String (SNOBOL4 with Pattern-Matching)
 - Should strings be simply a special kind of character array or a primitive type?
 - Should strings have static or dynamic length?
 - What operations? copy? slices? affect? length?
 - Notion of descriptor

- Numeric
- Boolean (since ALGOL 60)
- Characters
- String (SNOBOL4 with Pattern-Matching)
 - Should strings be simply a special kind of character array or a primitive type?
 - Should strings have static or dynamic length?
 - What operations? copy? slices? affect? length?
 - Notion of descriptor

Primitive types

- Numeric
- Boolean (since ALGOL 60)
- Characters
- String (SNOBOL4 with Pattern-Matching)
 - Should strings be simply a special kind of character array or a primitive type?
 - Should strings have static or dynamic length?
 - What operations? copy? slices? affect? length?
 - Notion of descriptor

Primitive types

- Numeric
- Boolean (since ALGOL 60)
- Characters
- String (SNOBOL4 with Pattern-Matching)
 - Should strings be simply a special kind of character array or a primitive type?
 - Should strings have static or dynamic length?
 - What operations? copy? slices? affect? length?
 - Notion of descriptor

1 What is Type Checking?

2 Why using Types?

3 The Concepts of Types

- Numeric Types
- Other Primitive Types
- **Complex Types**

4 (Crash) Introduction to Natural Deduction

Enumerations

- Integer ranges, or ordered labels

```
type days_num = 1 .. 7;
```

```
type days = (Monday, Tuesday, Wednesday, Thursday,  
            Friday, Saturday, Sunday);
```

- Are enumeration values coerced to integer?

- Conflicts are solved in Ada

```
type light is (red, orange, green);
```

```
type flag is (red, orange, green);
```

- Common operations

```
Pascal  
pred (orange)
```

```
Ada  
flag'pred(orange)
```

Enumerations

- Integer ranges, or ordered labels

```
type days_num = 1 .. 7;
```

```
type days = (Monday, Tuesday, Wednesday, Thursday,  
            Friday, Saturday, Sunday);
```

- Are enumeration values coerced to integer?

- Conflicts are solved in Ada

```
type light is (red, orange, green);
```

```
type flag is (red, orange, green);
```

- Common operations

Pascal

```
pred (orange)
```

Ada

```
flag'pred(orange)
```

- Integer ranges, or ordered labels

```
type days_num = 1 .. 7;
```

```
type days = (Monday, Tuesday, Wednesday, Thursday,  
            Friday, Saturday, Sunday);
```

- Are enumeration values coerced to integer?

- Conflicts are solved in Ada

```
type light is (red, orange, green);
```

```
type flag is (red, orange, green);
```

- Common operations

Pascal
`pred (orange)`

Ada
`flag'pred(orange)`

- Integer ranges, or ordered labels

```
type days_num = 1 .. 7;
```

```
type days = (Monday, Tuesday, Wednesday, Thursday,  
            Friday, Saturday, Sunday);
```

- Are enumeration values coerced to integer?

- Conflicts are solved in Ada

```
type light is (red, orange, green);
```

```
type flag is (red, orange, green);
```

- Common operations

Pascal

```
pred (orange)
```

Ada

```
flag'pred(orange)
```

- Characters are enumerations in Ada, contrary to Pascal, C, etc.

```
type digits is ('0', '1', '2', '3', '4',  
               '5', '6', '7', '8', '9');  
type odigits is ('0', '2', '4', '6', '8');
```

- Iteration over enumerations

```
for l in '0' .. '8' loop
```

- Disambiguation

```
for l in odigits('0')..odigits('8') loop
```

- Characters are enumerations in Ada, contrary to Pascal, C, etc.

```
type digits is ('0', '1', '2', '3', '4',  
               '5', '6', '7', '8', '9');  
type odigits is ('0', '2', '4', '6', '8');
```

- Iteration over enumerations

```
for l in '0' .. '8' loop
```

- Disambiguation

```
for l in odigits('0')..odigits('8') loop
```

- Characters are enumerations in Ada, contrary to Pascal, C, etc.

```
type digits is ('0', '1', '2', '3', '4',  
               '5', '6', '7', '8', '9');  
type odigits is ('0', '2', '4', '6', '8');
```

- Iteration over enumerations

```
for l in '0' .. '8' loop
```

- Disambiguation

```
for l in odigits'('0')..odigits'('8') loop
```


- Available since Autocodes.

- In Fortran [Sun Microsystems, 1996]:

up to 7 dimensions. Since FORTRAN IV, originally 3

```
REAL TAD(2,2,3,4,5,6,10)
```

Fortran 2008 requires supports up to 15.

free lower bounds

```
REAL A(3:5, 7, 3:5), B(0:2)
```

character arrays

```
CHARACTER M(3,4)*7, V(9)*4
```

- Available since Autocodes.
- In Fortran [Sun Microsystems, 1996]:

up to 7 dimensions Since FORTRAN IV, originally 3

```
REAL TAO(2,2,3,4,5,6,10)
```

Fortran 2008 requires supports up to 15.

free lower bounds

```
REAL A(3:5, 7, 3:5), B(0:2)
```

character arrays

```
CHARACTER M(3,4)*7, V(9)*4
```

- Available since Autocodes.
 - In Fortran [Sun Microsystems, 1996]:
 - up to 7 dimensions Since FORTRAN IV, originally 3
- ```
REAL TAO(2,2,3,4,5,6,10)
```
- Fortran 2008 requires supports up to 15.

free lower bounds

```
REAL A(3:5, 7, 3:5), B(0:2)
```

character arrays

```
CHARACTER M(3,4)*7, V(9)*4
```

- Available since Autocodes.
- In Fortran [Sun Microsystems, 1996]:
  - up to 7 dimensions Since FORTRAN IV, originally 3  
`REAL TAO(2,2,3,4,5,6,10)`  
Fortran 2008 requires supports up to 15.
  - free lower bounds  
`REAL A(3:5, 7, 3:5), B(0:2)`
  - character arrays  
`CHARACTER M(3,4)*7, V(9)*4`

- Available since Autocodes.
- In Fortran [Sun Microsystems, 1996]:
  - up to 7 dimensions Since FORTRAN IV, originally 3  
`REAL TAO(2,2,3,4,5,6,10)`  
Fortran 2008 requires supports up to 15.
  - free lower bounds  
`REAL A(3:5, 7, 3:5), B(0:2)`
  - character arrays  
`CHARACTER M(3,4)*7, V(9)*4`

- In Fortran [Sun Microsystems, 1996]:

```
CHARACTER BUF(10)
```

- C

```
char buf[10];
```

- Pascal

```
var buf : array [0 .. 9] of char;
```

- Ada

```
buf : array (0..9) of characters;
```

- C++ (2011)

```
std::array<char, 10> buf;
```

- In Fortran [Sun Microsystems, 1996]:  
`CHARACTER BUF(10)`
- C  
`char buf[10];`
- Pascal  
`var buf : array [0 .. 9] of char;`
- Ada  
`buf : array (0..9) of characters;`
- C++ (2011)  
`std::array<char, 10> buf;`

- In Fortran [Sun Microsystems, 1996]:  
`CHARACTER BUF(10)`
- C  
`char buf[10];`
- Pascal  
`var buf : array [0 .. 9] of char;`
- Ada  
`buf : array (0..9) of characters;`
- C++ (2011)  
`std::array<char, 10> buf;`



- In Fortran [Sun Microsystems, 1996]:  
`CHARACTER BUF(10)`
- C  
`char buf[10];`
- Pascal  
`var buf : array [0 .. 9] of char;`
- Ada  
`buf : array (0..9) of characters;`
- C++ (2011)  
`std::array<char, 10> buf;`

- In Fortran [Sun Microsystems, 1996]:  
`CHARACTER BUF(10)`
- C  
`char buf[10];`
- Pascal  
`var buf : array [0 .. 9] of char;`
- Ada  
`buf : array (0..9) of characters;`
- C++ (2011)  
`std::array<char, 10> buf;`

# Initializing Arrays [Sebesta, 2002]

- In Fortran:

```
INTEGER LIST(3)
DATA LIST /0, 3, 5/
```

- Ada

```
list : array (0..2) of INTEGER := (1, 3, 5);
conv : array (0..9) of INTEGER := (4 => 2,
 5 => 1,
 others => 0);
```

# Initializing Arrays

- C

```
int list[] = {42, 51, 96};
const char *const names[] = {"Foo", "Bar", "Baz", "Qux"};
```

- C99

```
int a[5] = { [3] = 29, 30, [1] = 15 }; // { 0, 15, 0, 29, 30 }
int whitespace[256]
 = { [' '] = 1, ['\t'] = 1, ['\h'] = 1,
 ['\f'] = 1, ['\n'] = 1, ['\r'] = 1 };
```

- GNU C

```
int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };
```

- C++ (2011)

```
std::array<int, 3> a1{ {1,2,3} }; // double-braces required
std::array<int, 3> a2 = {1, 2, 3}; // except after =
auto a3 = std::array<int, 3>{1, 2, 3};
```

# Operation on Arrays: APL [Sebesta, 2002]

$\phi V$  Reverse  $V$

$\phi M$  Reverse the columns of  $M$

$\theta M$  Reverse the rows of  $M$

$\emptyset M$  Transpose  $M$   
(symbol should be flipped vertically).

$\boxed{\div} M$  Invert  $M$

# Operation on Arrays: APL [Sebesta, 2002]

$\phi V$  Reverse  $V$

$\phi M$  Reverse the columns of  $M$

$\theta M$  Reverse the rows of  $M$

$\emptyset M$  Transpose  $M$   
(symbol should be flipped vertically).

$\boxed{\div} M$  Invert  $M$

# Operation on Arrays: APL [Sebesta, 2002]

$\phi V$  Reverse  $V$

$\phi M$  Reverse the columns of  $M$

$\theta M$  Reverse the rows of  $M$

$\emptyset M$  Transpose  $M$   
(symbol should be flipped vertically).

$\boxed{\div} M$  Invert  $M$

# Operation on Arrays: APL [Sebesta, 2002]

$\phi V$  Reverse  $V$

$\phi M$  Reverse the columns of  $M$

$\theta M$  Reverse the rows of  $M$

$\emptyset M$  Transpose  $M$   
(symbol should be flipped vertically).

$\boxed{\div} M$  Invert  $M$



# Operation on Arrays: APL [Sebesta, 2002]

$\phi V$  Reverse  $V$

$\phi M$  Reverse the columns of  $M$

$\theta M$  Reverse the rows of  $M$

$\emptyset M$  Transpose  $M$   
(symbol should be flipped vertically).

$\boxed{\div} M$  Invert  $M$

- Originally, references (no `&`, no arithmetics etc.).
- Dynamic memory allocation.

A C “feature”.

```
strlen.c
```

```
size_t strlen(const char *cp)
{
 size_t res = 0;
 for (; *cp; ++cp)
 ++res;
 return res;
}
```

# A Better strlen

## strlen.c

```
size_t strlen(const char *cp)
{
 const char *cp2 = cp;
 for (/* nothing. */; *cp2; ++cp2)
 continue;
 return cp2 - cp;
}
```

# An Even Better (?) strlen

```
size_t strlen(const char *str)
{
 size_t res = 0;
 for (uint32_t *i = (uint32_t *)str;; ++i)
 {
 if (!(*i & 0x000000ff)) return res;
 if (!(*i & 0x0000ff00)) return res + 1;
 if (!(*i & 0x00ff0000)) return res + 2;
 if (!(*i & 0xff000000)) return res + 3;
 res += 4;
 }
}
```

- Beware of endianness (here, little endian)
- Beware of alignment
- Beware of Valgrind
- Still four if, one can suffice

# An Even Better (?) strlen

```
size_t strlen(const char *str)
{
 size_t res = 0;
 for (uint32_t *i = (uint32_t *)str;; ++i)
 {
 if (!(*i & 0x000000ff)) return res;
 if (!(*i & 0x0000ff00)) return res + 1;
 if (!(*i & 0x00ff0000)) return res + 2;
 if (!(*i & 0xff000000)) return res + 3;
 res += 4;
 }
}
```

- Beware of endianness (here, little endian)
- Beware of alignment
- Beware of Valgrind
- Still four if, one can suffice

# An Even Better (?) strlen

```
size_t strlen(const char *str)
{
 size_t res = 0;
 for (uint32_t *i = (uint32_t *)str;; ++i)
 {
 if (!(*i & 0x000000ff)) return res;
 if (!(*i & 0x0000ff00)) return res + 1;
 if (!(*i & 0x00ff0000)) return res + 2;
 if (!(*i & 0xff000000)) return res + 3;
 res += 4;
 }
}
```

- Beware of endianness (here, little endian)
- Beware of alignment
- Beware of Valgrind
- Still four *if*, one can suffice

# An Even Better (?) strlen

```
size_t strlen(const char *str)
{
 size_t res = 0;
 for (uint32_t *i = (uint32_t *)str;; ++i)
 {
 if (!(*i & 0x000000ff)) return res;
 if (!(*i & 0x0000ff00)) return res + 1;
 if (!(*i & 0x00ff0000)) return res + 2;
 if (!(*i & 0xff000000)) return res + 3;
 res += 4;
 }
}
```

- Beware of endianness (here, little endian)
- Beware of alignment
- Beware of Valgrind
- Still four *if*, one can suffice



# An Even Better (?) strlen

```
size_t strlen(const char *str)
{
 size_t res = 0;
 for (uint32_t *i = (uint32_t *)str;; ++i)
 {
 if (!(*i & 0x000000ff)) return res;
 if (!(*i & 0x0000ff00)) return res + 1;
 if (!(*i & 0x00ff0000)) return res + 2;
 if (!(*i & 0xff000000)) return res + 3;
 res += 4;
 }
}
```

- Beware of endianness (here, little endian)
- Beware of alignment
- Beware of Valgrind
- Still four `if`, one can suffice

# GLibc's strlen

```
/* One character at a time, until aligned on longword. */
...
unsigned long int longword_ptr = (unsigned long int *) char_ptr;
unsigned long int himagic = 0x80808080L;
unsigned long int lomagic = 0x01010101L;
if (sizeof (unsigned long int) > 4) {
 himagic = ((himagic << 16) << 16) | himagic;
 lomagic = ((lomagic << 16) << 16) | lomagic;
}
for (;;) {
 unsigned long int longword = *longword_ptr++;
 if (((longword - lomagic) & ~longword & himagic) != 0) {
 /* Which of the bytes was the zero? If none of them were,
 it was a misfire; continue the search. */
 const char *cp = (const char *) (longword_ptr - 1);
 if (cp[0] == 0) return cp - str;
 ...
 }
}
```

# Pointers and Arrays in C

- Arrays are almost pointers in C
- $a[i] \rightsquigarrow *(a + i)$  (pointer arithmetic)
- $a + i = i + a$
- $a + i \rightsquigarrow a + i * \text{sizeof} (*a)$  (integer arithmetic)

## array[index] vs. index[array]

```
#include <stdio.h>

int main()
{
 int foo[2] = { 51, 42 };
 int zero = 0;
 printf("%d, %d\n", zero[foo], 1[foo]);
 return 0;
}
```

# Pointers and Arrays in C

- Arrays are almost pointers in C
- $a[i] \rightsquigarrow *(a + i)$  (pointer arithmetic)
- $a + i = i + a$
- $a + i \rightsquigarrow a + i * \text{sizeof} (*a)$  (integer arithmetic)

## array[index] vs. index[array]

```
#include <stdio.h>

int main()
{
 int foo[2] = { 51, 42 };
 int zero = 0;
 printf("%d, %d\n", zero[foo], 1[foo]);
 return 0;
}
```

# Pointers and Arrays in C

- Arrays are almost pointers in C
- $a[i] \rightsquigarrow *(a + i)$  (pointer arithmetic)
- $a + i = i + a$
- $a + i \rightsquigarrow a + i * \text{sizeof} (*a)$  (integer arithmetic)

array[index] vs. index[array]

```
#include <stdio.h>

int main()
{
 int foo[2] = { 51, 42 };
 int zero = 0;
 printf("%d, %d\n", zero[foo], 1[foo]);
 return 0;
}
```

# Pointers and Arrays in C

- Arrays are almost pointers in C
- $a[i] \rightsquigarrow *(a + i)$  (pointer arithmetic)
- $a + i = i + a$
- $a + i \rightsquigarrow a + i * \text{sizeof} (*a)$  (integer arithmetic)

array[index] vs. index[array]

```
#include <stdio.h>

int main()
{
 int foo[2] = { 51, 42 };
 int zero = 0;
 printf("%d, %d\n", zero[foo], 1[foo]);
 return 0;
}
```

# Pointers and Arrays in C

- Arrays are almost pointers in C
- $a[i] \rightsquigarrow *(a + i)$  (pointer arithmetic)
- $a + i = i + a$
- $a + i \rightsquigarrow a + i * \text{sizeof} (*a)$  (integer arithmetic)

## array[index] vs. index[array]

```
#include <stdio.h>

int main()
{
 int foo[2] = { 51, 42 };
 int zero = 0;
 printf("%d, %d\n", zero[foo], 1[foo]);
 return 0;
}
```

# Dynamic Arrays

- Difficult implementation: dynamic stack frame.
- Deferred to a later lecture dedicated to stack frames.



Typical of functional languages.

```
pair.hs
```

```
fst :: (a,b) -> a
```

```
fst (x,y) = x
```

```
snd :: (a,b) -> b
```

```
snd (x,y) = y
```

```
let type person =
 {
 first_name : string,
 last_name : string
 };
```

Inheritance/extension.

# Variants in Functional Language

Extremely useful for syntax trees [Anisko, 2003].

```
ast.hs
```

```
data Exp = Const Int
 | Name String
 | Temp String
 | Binop { op :: Op, lhs :: Exp, rhs :: Exp }
 | Mem Exp
 | Call { fun :: Exp, arg :: [Exp] }
 | ESeq { stm :: Stm, exp :: Exp }
data Op = Add | Sub | Mul | Div | Mod | And | Or
```

# Variants in Imperative Language

Decent in Pascal.

```
variants.p [gpc, 2003]
```

```
type
```

```
 EyeColorType = (Red, Green, Blue, Brown, Pink);
```

```
 PersonRec = record
```

```
 Age: Integer;
```

```
 case EyeColor: EyeColorType of
```

```
 Red, Green : (WearsGlasses: Boolean);
```

```
 Blue, Brown: (LengthOfLashes: Integer);
```

```
end;
```

# Unions

Sort of raw variants.

shape.c

```
typedef enum shape_kind_e
{
 shape_square,
 shape_circle,
 ...
} shape_kind_t;
```

```
struct Circle;
struct Square;
...
```

```
typedef struct Shape
{
 shape_kind_t kind;
 union {
 Square s;
 Circle c;
 ...
 } u;
} Shape;
```

- Algol has function types.
- Additional meaning in high-order functional languages.

# Tuples and Functions

`curry` convert an uncurried function to a curried function

`uncurry` convert a curried function to a function on pairs

## `curry.hs`

```
curry :: ((a, b) -> c) -> a -> b -> c
```

```
curry f x y = f (x, y)
```

```
uncurry :: (a -> b -> c) -> ((a, b) -> c)
```

```
uncurry f p = f (fst p) (snd p)
```

- A means to define a *list of X*, where  $X$  is a type variable.



# Genericity in C++

shape.cc

```
template <typename C>
class Shape
{
public:
 Shape(C x, C y)
 : x_(x), y_(y)
 {}
 // ...
private:
 C x_, y_;
};

auto f = Shape<int>{1, 2};
```

# Genericity in Eiffel

```
shape.e
```

```
class SHAPE[COORDINATE]
```

```
feature
```

```
 xc, yc : COORDINATE ;
```

```
 set_x_y(x,y : COORDINATE) is
```

```
 do
```

```
 xc := x ;
```

```
 yc := y ;
```

```
 end ;
```

```
...
```

# Genericity in Eiffel

```
shape.e
```

```
class SHAPE [COORDINATE->NUMERIC]
```

```
feature
```

```
 xc, yc : COORDINATE ;
```

```
 set_x_y(x,y : COORDINATE) is
```

```
 do
```

```
 xc := x ;
```

```
 yc := y ;
```

```
 end ;
```

```
...
```

# Genericity in Haskell: Signature of Maybe

## Maybe.hs

```
module Maybe(
 isJust, isNothing,
 fromJust, fromMaybe, listToMaybe, maybeToList,
 catMaybes, mapMaybe,
 -- ...and what the Prelude exports
 Maybe(Nothing, Just),
 maybe) where

data Maybe a = Nothing | Just a

isJust, isNothing :: Maybe a -> Bool
fromJust :: Maybe a -> a
fromMaybe :: a -> Maybe a -> a
listToMaybe :: [a] -> Maybe a
maybeToList :: Maybe a -> [a]
catMaybes :: [Maybe a] -> [a]
mapMaybe :: (a -> Maybe b) -> [a] -> [b]
```

# Genericity in Haskell: Implementation of Maybe

## Maybe.hs

```
isJust :: Maybe a -> Bool
isJust (Just a) = True
isJust Nothing = False

isNothing :: Maybe a -> Bool
isNothing = not . isJust

fromJust :: Maybe a -> a
fromJust (Just a) = a
fromJust Nothing = error "Maybe.fromJust: Nothing"

fromMaybe :: a -> Maybe a -> a
fromMaybe d Nothing = d
fromMaybe d (Just a) = a
```

# Genericity in Haskell: Implementation of Maybe

## Maybe.hs

```
maybeToList :: Maybe a -> [a]
maybeToList Nothing = []
maybeToList (Just a) = [a]

listToMaybe :: [a] -> Maybe a
listToMaybe [] = Nothing
listToMaybe (a:_) = Just a

catMaybes :: [Maybe a] -> [a]
catMaybes ms = [m | Just m <- ms]

mapMaybe :: (a -> Maybe b) -> [a] -> [b]
mapMaybe f = catMaybes . map f
```

## Tiger Types [Appel, 1998]

```
⟨Type⟩ ::= "Int" | "String" | "Void" | "Nil"
 | "Array" ⟨Type⟩
 | "Record" ("Id" ⟨Type⟩)
 | "Class" ("Id" ⟨Type⟩)
 | "Function" ("Id" ⟨Type⟩)
 | "Method" ("Id" ⟨Type⟩)
 | "Name" ⟨String⟩
```

# Comparing two Types

## Equivalence

- by structure
- by name

Pascal report did not define “equivalent types”.

```
equivalence.p
```

```
type link = ^cell;
var next : link;
 last : link;
 p : ^cell;
 q, r : ^cell;
```



# Comparing two Types for Identity

equivalence.tig

```
type original = { value : int }
type copy = { value : int }
type alias = original
```

```
compatibility.ada
```

```
subtype index1 is integer 1 .. 10;
type index2 is new integer 1 .. 10;
```

Subtypes are constrained types.

# (Crash) Introduction to Natural Deduction

1 What is Type Checking?

2 Why using Types?

3 The Concepts of Types

4 (Crash) Introduction to Natural Deduction

# Natural Deduction for Intuitionistic Logic

$$\frac{}{A_1, \dots, A_n \vdash A_k}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}$$

$$\frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

# Natural Deduction for Intuitionistic Logic

$$\frac{}{A_1, \dots, A_n \vdash A_k}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}$$

$$\frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

# Natural Deduction for Intuitionistic Logic

$$\frac{}{A_1, \dots, A_n \vdash A_k}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}$$

$$\frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

# Natural Deduction for Intuitionistic Logic

$$\frac{}{A_1, \dots, A_n \vdash A_k}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}$$

$$\frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$

# Natural Deduction for Intuitionistic Logic

$$\frac{}{A_1, \dots, A_n \vdash A_k}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \times B} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash A} \quad \frac{\Gamma \vdash A \times B}{\Gamma \vdash B}$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \quad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A + B} \quad \frac{\Gamma \vdash B}{\Gamma \vdash A + B}$$

$$\frac{\Gamma \vdash A + B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}$$



# Natural Deduction for Intuitionistic Logic

$$\frac{}{x_1 : A_1, \dots, x_n : A_n \vdash x_k : A_k}$$

$$\frac{\Gamma \vdash u : A \quad \Gamma \vdash v : B}{\Gamma \vdash (u, v) : A \times B} \quad \frac{\Gamma \vdash u : A \times B}{\Gamma \vdash \text{fst}(u) : A} \quad \frac{\Gamma \vdash u : A \times B}{\Gamma \vdash \text{snd}(u) : B}$$

$$\frac{\Gamma, x : A \vdash u : B}{\Gamma \vdash \lambda x^A. u : A \Rightarrow B} \quad \frac{\Gamma \vdash f : A \Rightarrow B \quad \Gamma \vdash u : A}{\Gamma \vdash fu : B}$$

$$\frac{\Gamma \vdash u : A}{\Gamma \vdash \text{inl}^{A+B}(u) : A + B} \quad \frac{\Gamma \vdash u : B}{\Gamma \vdash \text{inr}^{A+B}(u) : A + B}$$

$$\frac{\Gamma \vdash w : A + B \quad \Gamma, x : A \vdash u : C \quad \Gamma, y : B \vdash v : C}{\Gamma \vdash \text{case } w \text{ of } \text{inl}(x).u \mid \text{inr}(y).v : C}$$

$$\frac{}{\Gamma \vdash n : \text{Int}} \quad \frac{}{\Gamma \vdash s : \text{String}}$$
$$\frac{}{x_1 : A_1, \dots, x_n : A_n \vdash x_k : A_k}$$
$$\frac{\Gamma \vdash a : \text{Int} \quad \Gamma \vdash b : \text{Int}}{\Gamma \vdash a + b : \text{Int}} + \quad \dots$$

$$\frac{\Gamma \vdash c : \text{Int} \quad \Gamma \vdash t : A \quad \Gamma \vdash f : A}{\Gamma \vdash \text{if } c \text{ then } t \text{ else } f : A} \text{ if then else}$$
$$\frac{\Gamma \vdash c : \text{Int} \quad \Gamma \vdash t : \text{Void}}{\Gamma \vdash \text{if } c \text{ then } t : \text{Void}} \text{ if then}$$

# Bibliography I



(2003).

*The GNU Pascal Compiler Manual.*

Free Software Foundation, 59 Temple Place – Suite 330, Boston, MA 02111-1307 USA, 20030830 edition.



Anisko, R. (2003).

Havm.

<http://tiger.lrde.epita.fr/Havm>.



ANSI/IEEE (1987).

ANSI/IEEE std 854-1987 – IEEE standard for radix-independent floating-point arithmetic.



Appel, A. W. (1998).

*Modern Compiler Implementation in C, Java, ML.*

Cambridge University Press.



Goldberg, D. (1991).

What every computer scientist should know about floating-point arithmetic.

*ACM Computing Surveys (CSUR)*, 23(1):5–48.

[http://docs.sun.com/source/806-3568/ncg\\_goldberg.html](http://docs.sun.com/source/806-3568/ncg_goldberg.html).



Sebesta, R. W. (2002).

*Concepts of programming languages (5th ed.)*.

Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA.



Sun Microsystems (1996).

Fortran 77 Language Reference (for f77 4.2).

<http://www.ictp.trieste.it/~manuals/programming/sun/fortran/f77rm/index.html>.