3 Values and types

3.1 Value systems

Every PL manipulates values. But, what are values? Intuitively, a value is...

- an “entity” that exists during computation
- anything that may be manipulated by a program
- anything that may be passed as an argument to a Function or Procedure (in Pascal)

Definition 3.1.1 (The values’ universe).

- Every PL has a set of values, e.g., integers, tuples, records, functions,...
- Running a program of the PL, amounts to manipulation of members of this set.
- The values’ set is also called the universe of values.

3.1.1 Symbolic values

3.1.2 Semantics of $S$-expressions

3.1.3 Expressions

3.1.4 Function application

3.1.5 Control structures

3.1.6 Lambda abstraction

3.1.7 Higher-order functions

3.1.8 Recursion

3.1.9 Structural induction

3.1.10 Type systems

3.1.11 Summary

3.2 Introduction to types

3.2.1 Types as sets

3.2.2 Type theory

3.2.3 Typed determines semantics: the overloading case study

3.2.4 Summary

3.3 The type constructors of MOCK

3.3.1 Value systems

3.3.2 Power sets

3.3.3 Product type constructor

3.3.4 Branding

3.3.5 Union and choice/disjoint union type constructors

3.3.6 Records

3.3.7 Special types: Unit, Top & Bottom

3.3.8 Mapping as functions and arrays

3.3.9 Typeclasses

3.3.10 Recursive types

3.3.11 Summary

3.4 Type constructors in actual PLs

3.4.1 Product type constructor

3.4.2 Integral exponentiation

3.4.3 Branding

3.4.4 Union and choice/disjoint union type constructors

3.4.5 Tags in concrete PLs

3.4.6 $\mu$-LISP in C

Exercises

References
3. Values vs. variables

A value is not a variable:
- A variable may store a value.
- A variable may also be undefined.
- Some PLs (e.g., ML) do not offer variables.

Why the confusion?
- In traditional imperative PLs, it is difficult to construct complex, interesting values, without using variables.
- People familiar with these PLs, tend to think of integers, reals, etc., as values, but they may have hard time understanding that there are also array values, function values, etc.

4. Machine representation vs. types vs. values

A type system is a set of types. A type is a set of values:
- Subsets are not necessarily disjoint.
- Subsets are not necessarily different.

Machine representation: mapping of an element in \( \mathcal{V} \) to the machine.
- \( \mathcal{V}_L \) the set of all values of a language \( \mathcal{L} \), AKA the values' universe.
- \( \mathbb{T}_\mathcal{L} \) the type system of \( \mathcal{L} \) (with respect to \( \wp \mathcal{V}_L \))

\[
\mathbb{T}_\mathcal{L} \subseteq \wp \mathcal{V}_L \quad (3.1.1)
\]

\( \mathcal{P}_\mathcal{L} \) policy for representation of values of \( \mathcal{L} \) on different machines \( M_1, M_2, \ldots \),

\[
\mathcal{P}_\mathcal{L} : \mathcal{V}_\mathcal{L} \rightarrow \{M_1, M_2, \ldots\} \quad (3.1.2)
\]

5. Value structure

- Atomic value: is not composed of other values
  - truth values, characters, integers, reals, pointers
- Composite value: is composed of other values
  - records, arrays, sets, files

The ways to create composite values in a PLs are usually independent of its implementation.

The set of legal values in a PL’s implementation:
- a closure of the atomic values in this implementation under the mechanisms the PL specification allows for creating composite values

6. Universes including very simple values

To emphasize the difference, we start with simple, yet very useful value systems:

- LISP
- MATHEMATICA
- PROLOG

In essence, in these PLs, all values are “symbolic expressions”. There are no types in any of these universes.

7. \( \mathcal{V}_{\text{Lisp}} = S \)-expressions

An \( S \)-Expression\(^5\) is
1. An Atom
2. \((S_1, S_2)\), where \( S_1 \) and \( S_2 \) are \( S \)-expressions

An Atom is
1. A string of any length (Many LISP implementations ignore letter case)
2. The special value NIL.
3. An integer (non-essential)
4. A real number (non-essential)

Examples:
- \( \text{hello} \)
- \( (a, \text{NIL}) \)
- \( (\text{NIL}, a) \)
- \( (\text{Hello}, \text{(war, lord)}) \)

8. \( S \)-expression as binary trees

- \( S \)-expressions are binary trees whose leaves are either string or NIL.
- LISP “assumes”\(^7\) that they are indeed represented as trees:
  - cons Internal nodes are called CONS nodes.
  - car Left pointer
  - cdr Right pointer

---

\( ^2 \)Later, we will see that it is not just any set of values.
\( ^3 \)For a set \( S \), the notation \( \wp S \) stands for the power set of set \( S \), i.e., \( \wp S = \{ S | S \subseteq S \} \).
\( ^4 \)Here, and henceforth, PLs = Programming Languages.

---

\( ^5 \)LISP is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use LISP itself a lot.” Eric Raymond, “How to Become a Hacker”

\( ^6 \)\( S \)-Expressions are symbolic expressions LISP style.

\( ^7 \)actual representation could be different.

---

Figure 3.1.1: The CONS record
9. Examples of S-expression as binary trees

<table>
<thead>
<tr>
<th>S-expression</th>
<th>Binary Tree</th>
</tr>
</thead>
<tbody>
<tr>
<td>a.NIL</td>
<td>![Binary Tree of a.NIL]</td>
</tr>
<tr>
<td>NIL a</td>
<td>![Binary Tree of NIL a]</td>
</tr>
<tr>
<td>a a</td>
<td>![Binary Tree of a a]</td>
</tr>
</tbody>
</table>

Hello.(var.lord)

10. List shorthand

The list

(a b c d)

is shorthand (syntactic sugar) for

(a. (b. (c. (d. NIL))))

In binary tree representation:

![Binary Tree of (a b c d)]

Take note that not all S-expressions can be written using the list notation.

11. Quiz

1. What does () mean in the tree notation?
2. What does ((a b) (c d)) mean in the tree notation?
3. Can you give an example of an S-expression which cannot be represented using the list notation?

3.1.2 Semantics of S-expressions?

7 Frames: □ Visualizing the type system □ What’s in a type system? □ Value–type association □ Types look like sets of values □ Not all sets of values make a type □ Type equality vs. set equality □ Type systems are defined recursively □ Significance of type □ Purpose of type? □ How does the PL/compiler/runtime uses types? □ Type error □ Type error on multiple values □ Dealing with type errors □ Dealing with pseudo type error □ Dealing with other type errors □ Case study: Gnu-C-Compiler handling of division by zero □ Types & underlying machine □ Opacity of machine representation?

12. “Semantics” of the list notation

The evaluation of a list

ℓ = (a b c d)

means

apply function car(ℓ) to the list of arguments cdr(ℓ)

Example

Lisp

> (+ 2 (* 3 4))

14

What does it mean to “evaluate”?

Atom Find out the “definition” of this atom in the symbol table.

Literal Itself

List Recursively evaluate all list elements, and then apply the first argument as a function to the remaining arguments.

13. Should you care to run Lisp?

Installing and running Gnu-Lisp:

The program ‘gcl’ is currently not installed. You can install it by typing:

`sudo apt-get install gcl`

% sudo apt install gcl

Reading package lists…

Building dependency tree…

The following NEW packages will be installed:

gcl

0 upgraded, 1 newly installed, 0 to remove and 38 not upgraded

% gcl

GCL (GNU Common Lisp) 2.6.7 CLtL1 Jul 27 2013 12:54:39

Source License: LGPL(gcl,gmp), GPL(unexec,bfd,xgcl)

> ^D

Observe that

• At first, Gnu-Lisp was not installed.
• User does as instructed and installs it
• User runs Gnu-Lisp
• User hits Ctrl-D at the prompt to exit

14. Interpreters: read evaluate print loop (REPL)

All interpreters follow the same scheme

1. Read input (and parse it)
2. Evaluate (call function eval in Lisp)
3. Print the result
4. Loop

15. Demonstrating lists’ semantics in Lisp

% gcl
> (a b c d)
Error: The function A is undefined.
Fast links are on: do (si::use-fast-links nil) for debugging
Error signaled by EVAL.
Broken at SYSTEM::GCL-TOP-LEVEL. Type :H for Help.

16. The four most basic Lisp functions

(quote γ) Do not evaluate γ
(car γ) First element of the list γ
(cdr γ) The rest of the list γ (everything but (car γ))
(cons γ δ) The list whose car is γ and whose cdr is δ

Using car and cdr

> (car (a b))
Error: The function A is undefined
> (car 'a b)
A
> cdr 'a b)
B
> (cons 'a b) '(c d)
((a B) C D)

17. The car/cdr notation

CAR
• Contents of the Address part of Register number
• Also called in other PLs:
  – “first”
  – “head” (or “hd” for short)

CDR
• Contents of the Decrement part of Register number
• Also called in other PLs:
  – “rest”
  – “tail” (or “tl” for short)

18. The “c[ad]+r” syntactic sugar

For brevity, let’s use the binding of the name z to the value ((a b) (c d)), then, i.e., evaluate

(setq 'z '((a b) c d))

then,

<table>
<thead>
<tr>
<th>LONG FORM</th>
<th>SHORT FORM</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>(car (car z))</td>
<td>(caar z)</td>
<td>A</td>
</tr>
<tr>
<td>(cdr (car z))</td>
<td>(cadar z)</td>
<td>B</td>
</tr>
<tr>
<td>(car (cdr z))</td>
<td>(cadr z)</td>
<td>C</td>
</tr>
<tr>
<td>(cdr (cdr z))</td>
<td>(caddr z)</td>
<td>D</td>
</tr>
</tbody>
</table>

Table 3.1.1: Examples of using the “c[ad]+r” syntactic sugar

19. Understanding “c[ad]+r” syntactic sugar

20. Names and literals in Lisp

The set function

> (set a b)
Error: The variable A is unbound.
> (set 'a b)
Error: The variable B is unbound.
> (set 'a 'b)
B
> a
B
> (set a 3)
3
> a
B
> (print a)
B
B

21. “List” & “if”

List:

(list 'a 'b 'c) = (a b c)
(list 'a 'b (list 'c 'd)) = (a b (c d))
(list (list 'a 'b) (list 'c 'd)) = ((a b) (c d))

If:

(if 'x 'a 'b) = a
(if (list 'x) 'a 'b) = a
(if '(x y) 'a 'b) = a
(if NIL 'a 'b) = b
(if () 'a 'b) = b
22. \(\lambda\)-functions

Function “lambda” makes it possible to define anonymous functions:

<table>
<thead>
<tr>
<th>Defining a (\lambda)-function</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>(lambda (x) (cons (cdr x) (car x)))</code></td>
</tr>
<tr>
<td><code>(LAMBDA-CLOSURE (\ x) (CONS (CDR X) (CAR X)))</code></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Defining and applying a (\lambda)-function:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>((lambda (x) (cons (cdr x) (car x))) '((a b) (c d)))</code></td>
</tr>
<tr>
<td><code>(((C D)) A B)</code></td>
</tr>
</tbody>
</table>

23. Named functions

- Defining
  
  ```lisp
  (defun foo (x) (cons (cdr x) (car x)))
  `FOO`
  ```

- Applying:
  
  ```lisp
  (foo '((a b) (c d)))
  `(((C D)) A B)`
  ```

24. Summary: Values in Lisp

- Extremely simple
- Have no types
- Simple basic operations: `car`, `cdr`, `cons`, `list`, `quote`, `if`, `defun`
- Can be used to compute anything! (conditions and recursion)
- The mother of all functional PLs.

25. Prolog intuitively

Simple: Almost the same as in Lisp\(^8\). Specifically:

- Trees of arbitrary degree:
  - Internal nodes carry label (unlike S-expressions)
  - Leaves are either
    - Labels
    - Symbolic “variables”

Only one fundamental operation on values: \(^9\)

Unification: Given two trees, replace (if possible) “variables” in each of them so that they become the same tree.

26. Prolog precisely

All values are terms. A term is either one of

- Atom a name with no inherent meaning.
- Number
- Variable must start with an upper case letter
- Composite which includes
  1. an atom called `functor`
  2. a list of any number of terms (arguments).

A list is a kind of term written as this `[a,B,c,x]`

27. Mathematica

Same as Prolog/LISP with lots of syntactic sugar:

![Figure 3.1.4: The many ways for presenting the symbolic values of Mathematica](image)

28. More traditional values

<table>
<thead>
<tr>
<th>Bash</th>
<th>Values of Bash</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Numbers</td>
<td></td>
</tr>
<tr>
<td>2. Strings</td>
<td></td>
</tr>
<tr>
<td>3. List of values</td>
<td></td>
</tr>
<tr>
<td>4. One dimensional arrays</td>
<td></td>
</tr>
</tbody>
</table>

---

\(^8\)recall the isomorphism between binary trees and forest of general trees
\(^9\)Come back to this slide at the end of the course
29. **VML: values in ML**

All the values in ML are first-class values:

**Atomic values** truth values, integers, reals, strings.

**Composite values** records, tuples (records w/o field names), constructions (tagged values), lists, arrays.

**Function values**

**References to variables**

What we can do in ML but not in Pascal:

- create a record composed of two functions
- write a function that gets a function \( f: \text{int} \to \text{int} \) and returns the composition of \( f \) with itself
- write an expression whose value is a reference to a variable

**30. Expressions**

**Definition 3.1.2 (Expression).** An expression is a part of a program whose evaluation during computation outcomes with a value.

- Several expressions in Pascal
  - `3.1416 chr(ord(‘%’)+1) ’Hello, world’ 2*a[i]+7`  
  - `sqr(4) q’.head`

**An expression in ML**

```
if leap(year) then 29 else 28
```

**31. Expressions are recursively defined**

Naturally, each PL is different, but the general scheme is:

Atomic expressions

- literals
- variable inspection

Expression constructors

- Operators such as “+”, “-”, ...
- Function calls

*The set of atomic expressions and the constructors’ set are PL dependent, but the variety is not huge.*

**32. Function call expression constructor**

**Static typing version**

Let \( f \) be a (typed) function of \( n \geq 0 \) arguments,

\[
f \in \tau_1 \times \cdots \times \tau_n \to \tau.
\]

Let \( E_1, \ldots, E_n \) be expressions of types \( \tau_1, \ldots, \tau_n \). Then, the call

\[
f(E_1, \ldots, E_n)
\]

is an expression of type \( \tau \).

**33. Functions vs. operators**

Both are:

- constructors of expressions
- apply an operation to values

Differences are largely syntactical

- Name
- Position, e.g., prefix or infix
- Parenthesis
- Precedence rules
- User overloading

**34. Syntactical differences between functions & operators**

<table>
<thead>
<tr>
<th></th>
<th>Functions</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Position?</td>
<td>prefix</td>
<td>prefix, infix, prefix</td>
</tr>
<tr>
<td>Precedence rules?</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>Parenthesis required?</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Name?</td>
<td>identifier</td>
<td></td>
</tr>
<tr>
<td>Arity?</td>
<td>0, 1, 2, 3,...</td>
<td>1: prefix/postfix operators 2: infix operators 3: C’s “? : ”</td>
</tr>
<tr>
<td>Overloadable by programmer?</td>
<td>Pascal, C: ✓</td>
<td>C++: ✓</td>
</tr>
</tbody>
</table>

Table 3.1.2: Syntactical differences between functions & operators

**References**

- CAR and CDR
- CONS
- defun
- lambda
- LISP
- MATHEMATICA
- S-Expressions
- Values of PROLOG
- Wolfram (PL used in MATHEMATICA)
3.2 Introduction to types

10 Frames: □ Riddle □ Unraveling the Marx riddle □ Context dependent resolution of overloading □ Keyword overloading in PASCAL □ Overloading □ Overloading in English □ “Plain” overloading vs. identifier/operator overloading □ Built-in procedure overloading in PASCAL □ Programmer defined overloading □ Function overloading in C++ □ Overloading the division operator in ADA □ Resolving ambiguity of overloading □ Ambiguity resolution

3.2.1 Types as sets

A PL \( \mathcal{L} \) has a set of values, \( V_\mathcal{L} \) with many values in it. A type is a set of these values, e.g.,
\[
|T_1| = 2.
\]

Types may be disjoint, e.g.,
\[
|T_1| \cap |T_2| = \emptyset.
\]
or contained, e.g.,
\[
T_1 \subseteq T_2.
\]

A type may be empty, e.g.,
\[
|T_3| = 0.
\]
a singleton, e.g.,
\[
|T_6| = 1.
\]
and even contain the entire set of values e.g.,
\[
T_7 = V_\mathcal{L}.
\]

Figure 3.2.1: Visualizing the type system

The type system is the set of all types; in our example,
\[
\mathcal{V} = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7\}
\]

36. What’s in a type system?

For a language \( \mathcal{L} \), let \( \mathbb{T}_\mathcal{L} \) denote its type system. Then,

- Each type is a subset of the values’ universe.
\[
\forall \mathbf{r} \in \mathbb{T}_\mathcal{L} : \mathbf{r} \subseteq V_\mathcal{L}.
\]

- \( \mathbb{T}_\mathcal{L} \) is a set of subsets of the values’ universe
\[
\mathbb{T}_\mathcal{L} \subset \wp(V_\mathcal{L}).
\]

- Types do not make a partition of \( V_\mathcal{L} \)
  - One type may be contained in another
  - The intersection of two types is not necessarily empty

37. Value–type association

For \( v \in V \), let
\[
\text{types}(v) = \{T \in \mathbb{T}_\mathcal{L} | v \in T\}
\]
Then,

- Every value has a type
\[
\forall v : \#\text{types}(v) > 0
\]

- Some values have more than one type,
\[
\exists v : \#\text{types}(v) > 1
\]

- Often
\[
\#\text{types}(v) > 1 \quad \Rightarrow \quad \#\text{types}(v) = \infty
\]
e.g., in C, “0” belongs to all pointer types
\[
T \in \mathbb{T}_C \Rightarrow “0” \in T^*
\]

38. Types look like sets of values

- Each type \( T \), defines a \( S_T \), the set of values of \( T \).
- Types describe values and expressions
- With every type there is “set predicate”:

Predicate on values A value \( v \) is of type \( T \), iff \( v \in S_T \)

Predicate on expressions An expression \( E \) belongs in type \( T \) iff every value \( v \) that \( E \) may evaluate to, satisfies \( v \in S_T \)

Shorthand for \( S_T \)? \( T = S_T \)?

- a common “abuse of notation” by which \( T \) sometimes means \( S_T \)
- get ready, everyone uses this abuse
39. Not all sets of values make a type

Every type “is” a set of values, but not every set of values “is” a type.

<table>
<thead>
<tr>
<th>Set of values</th>
<th>Is a type?</th>
</tr>
</thead>
<tbody>
<tr>
<td>{4.20, “Cannabis”, true, ‘?!’}</td>
<td>No (in most PLs)</td>
</tr>
<tr>
<td>{true, false}</td>
<td>Yes (in most PLs)</td>
</tr>
</tbody>
</table>

Table 3.2.1: A set of values which does not makes a type and a set that does

- Only sets which are atomic types, or constructed by the type constructors are types.
- Type $T$ also defines a set of allowed operations
- Conversely, all $v \in T$ must
  - Recognize the same operations
  - Respond similarly to each recognized operation

40. Type equality vs. set equality

Type equality: When is $T_1 = T_2$?
- If it holds for the corresponding sets that $T_1 = T_2$?
- Not always!
- Actually, the answer is most often negative!

Type containment: When is $T_1 \leq T_2$?
- Important, e.g., for assuring compatibility of actual to formal parameter.
- If it holds for the corresponding sets that $T_1 \subseteq T_2$?
- Not always!
- Actually, the answer is most often negative!

41. Type systems are defined recursively

Atomic types
- Make the recursion’s base.
- Mostly predefined by the PL.
- Programmer defined atomic types also happen.
- AKA: basic types or primitive types.

Type constructors
- Defined by the PL.
- Typically modeled after “theoretical type constructors” (Unit 3.3)
- May take 0, 1, 2, or any number of type arguments
- May take other arguments (e.g., integers or labels)

42. Significance of type

An expression using the $\cdot [\cdot]$ binary operator:
- Is it legal?
- What does it mean?
- What type is the result?

In C:

Legal if $i$ is a pointer and $x$ is an integer, e.g.,

```c
void f() {
    int *i = &x;
    x[i] = x ** i; // x
}
```

Illegal if $i$ is a floating point number and $x$ is a char,

```c
void f() {
    float i = 4.20;
    char x = ‘c’;
    x[i]; // x
}
```

type: subscripted value is neither array nor pointer nor vector

43. Purpose of type?

We have values; why do we need types?

Taxonomy of values; describe data effectively

Legality determine set of legal operations on values (prevent type errors, e.g., multiply a pointer by a set)

Semantics determine semantics of operations on values

Representation define program-machine interface:

Program $\Rightarrow$ Machine how to code values on different machines

Machine $\Rightarrow$ Program how to decode sequences of bits as actual values

44. How does the PL/compiler/runtime uses types?

Given $v \in V$ and an operation $\text{op}$

Legality use $\text{types}(v)$ to determine whether $\text{op}$ is applicable to $v$.

(Subunit 3.2.2 just ahead)

Semantics use $\text{types}(v)$ to determine the semantics of $\text{op}(v)$

(Subunit 3.2.3 just ahead)

Inference determine the type of $\text{op}(v)$

Representation (Unit 5.4)

Encode How to store $v$ in memory

Decode How to interpret the bits representing $v$
3.2.2 Type errors

39 Frames: □ Power sets □ Definition of type constructors is not enough □ Value constructors for power sets □ Defining operators on values of power sets □ Mock does not bother with practical issues

43. Type error

Example:

Type error in C

```c
int main(int argc, char *argv[]) {
    return argc / argv; // X
}
invalid operands to binary / (have 'int' and 'char **')
```

We can define:

**Definition 3.2.1** (Type error [on a single value]). A program commits a **type error** on a value \( v \in T \) if it makes an attempt to manipulate \( v \) in a way which is inconsistent with \( T \).

But, type errors can be committed on several values.

45. Type error on multiple values

**Definition 3.2.2** (Type error [on multiple values]). A program commits a **type error** on values \( v_1, \ldots, v_n \in T_1, \ldots, T_n \), \( n \geq 1 \) if it makes an attempt to manipulate \( v_1, \ldots, v_n \) in a way which is inconsistent with \( T_1, \ldots, T_n \).

In our example, the type error was committed on two values, \( v_1 = \text{argc} \) and \( v_2 = \text{argv} \).

Type error in C

```c
int main(int argc, char *argv[]) {
    return argc / argv; // X
}
invalid operands to binary / (have 'int' and 'char **')
```

47. Dealing with type errors

PLs report type errors

**Dynamically** When the program tries to commit them at runtime

**Statically** When it cannot be proved that the program will not commit them.

In the above case, the type error \( \text{argc} / \text{argv} \) was detected statically.

48. Pseudo type error

**Definition 3.2.3** (Pseudo type error). A program commits a **pseudo type error** on a value \( v \in T \) if it makes an attempt to manipulate \( v \) in a way

- which is, in general, legal for \( T \).
- yet, is illegal for the particular \( v \in T \).

Examples;

- Division by zero
- Array index overflow
- Add 100 to MAXINT in PASCAL
- Null pointer access

49. Dealing with pseudo type errors

**Statically** Impossible in general

- Think of \( \text{read}(i); \ a[i] := 0 \)
- Smart compilers can make early detection in some cases

**Dynamically** When the program tries to commit these at runtime

**Silently** No errors are generated if

- a PASCAL program adds 100 to MAXINT.
- a C program tries to access the 101st cell in an array with 100 cells.

50. Case study: Gnu-C-Compiler handling of division by zero

Gnu-C-Compiler often detects cases of division by zero

```c
int main() {
    return 0/(main-main); // X
}
warning: division by zero [-Wdiv-by-zero]
```

But, not all of them

```c
int f() { return 0 - 0; }
int main() {
    return 0/f(); //
}
```

Nevertheless, all division by zero errors are detected at runtime

51. Types & underlying machine

**Machine language** all values are untyped bit patterns

**Tagged architectures** add type information to values

- very rare
- no support for compound types

**Assembly language** minimal support for types, e.g., addresses vs. data

**High-level PLs** attach types to

- values
- expressions
- memory cells

52. Opacity of machine representation?

- All values are represented as a sequence of bits
- The type system often tries to hide this fact.

**Opaque** references in JAVA and ML are

Programmer have no way of telling how they are represented
**Transparent** e.g., `int` in Java is transparent
- it must be 32 bits wide
- it must use two’s complement

**Semi-transparent** Type `char` in C is semi-opaque:
- Must be at least 8 bits wide
- Cannot use more bits than `short`
- Cannot be `signed` or `unsigned`
- Can use one’s complement or two’s complement

the language tells you much, but not all, about the representation

---

**53. Type punning**

Revealing the mascaraed of bit sequences as values:

**Definition 3.2.4** (Type punning). *Type punning* is the power to interpret machine representation of `v` in a way which is inconsistent with `v`

**Type punning has the power to**

**Peep** into the bit sequence implementation of a type

**Type casting**

```c
long i, j;
int *p = &i, *q = &j;
long L1 = ((long)p);
long L2 = ((long)q);
long L = L1^L2;
```

**Mutilate** a value, by subjecting it to operations not allowed for its type

**Union type**

```c
union { double foo;
    long bar;
} baz;
baz.foo = 3.7;
printf("%ld\n", baz.bar);
```

---

**54. Type punning in C#**

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

```c
class Foo {
    public static void Main() {
        unsafe {
            // could also annotate class Foo
            // or function Main
            int i = 14;
            int *p = &i;
            Console.WriteLine("i = \$i\n", i);
            Console.WriteLine("i = \$i\n", *p);
            Console.WriteLine("p = \$p\n", p);
            Console.WriteLine("Address = \$r\n", &i);
        }
    }
    // unsafe block
    // function Main
    // class Foo
}
```

---

**3.2.3 Type determines semantics: the overloading case study**

4 Frames: \* Integral exponentiation \* Operators for integral exponentiation

---

**55. Riddle**

Can you figure out this?

**Groucho Marx (1890–1977):**

*Time flies like an arrow. Fruit flies like a banana!*  

---

**56. Unraveling the Marx riddle**

**Table 3.2.2: Overloaded meanings of terms in the Groucho Marx riddle**

<table>
<thead>
<tr>
<th>Term</th>
<th>Meaning I</th>
<th>Meaning II</th>
</tr>
</thead>
<tbody>
<tr>
<td>flies</td>
<td>flows</td>
<td>winged insects</td>
</tr>
<tr>
<td>like</td>
<td>similar to</td>
<td>favor, prefer</td>
</tr>
</tbody>
</table>

---

**57. Context dependent resolution of overloading**

**Figure 3.2.2: Unraveling the Marx riddle**

Time flies like an arrow. Fruit flies (the disgusting insects) like (favor) a banana!

---

**Figure 3.2.3: Archie Bunker explains to Edith Bunker how context is used to resolve the ambiguity of the three overloaded meanings of the word “Shalom” in Hebrew**

*Archie Is Branded*

https://www.youtube.com/watch?v=RdCIIEycwpS#t=8m56  
episode 57 (episode 20/season III) All In The Family 1973
60. Overloading in English

Unrelated meanings:

lie to present false information with the intention of deceiving

“I did not lie in the deposition”

lie to place oneself at rest in a horizontal position

“I did not lie in this position”

Close (more or less) meanings:

fly to move through the air

fly to travel by an airplane

fly a two-winged insect, such as insect

Lemma 3.2.6 (The fundamental rule of overloading). The intended meaning is figured out by context

61. “Plain” overloading vs. identifier/operator overloading

• Overloading of of in Pascal is not of an identifier.

• Keyword of does not name a nameable entity such as variable, a function, a procedure, etc.

Definition 3.2.7 (Identifier overloading). An identifier or operator is said to be overloaded if it simultaneously denotes two or more distinct nameables

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

• Integer addition

• Floating point addition

62. Built-in procedure overloading in Pascal

Output is

0
0.0000000000E+00
FALSE
Sunday

The identifier WriteLn in Pascal denotes many distinct functions.
63. Programmer defined overloading

- C does not allow operator overloading by programmer.
  
- PASCAL does not allow operator overloading by programmer.

64. Function overloading in C++

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

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

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

65. Overloading the division operator in Ada

- PASCAL forbids operator overloading by programmer, but, its younger, fatter, and uglier, “daughter”, Ada, allows it:

  Built-in semantics of “/”:

  Integer division \( \text{Integer} \times \text{Integer} \to \text{Integer} \)

  Real division \( \text{Real} \times \text{Real} \to \text{Real} \)

66. Resolving ambiguity of overloading

The actual meaning is determined by context:

(i) which parameters are passed to operator “/” upon invocation

(ii) how its result is used.

67. Ambiguity resolution

Consider the call \( \text{Id}(E) \) where \( \text{Id} \) denotes both:

- a function \( f_1 \) of type \( S_1 \to T_1 \)
- a function \( f_2 \) of type \( S_2 \to T_2 \)

Context Independent (C++)

- Either \( f_1 \) or \( f_2 \) is selected depending solely on the type of \( E \)
- We must have \( S_1 \neq S_2 \)
- May lead to ambiguities in certain cases, e.g., \#types > 1

Context Dependent (Ada)

- Either \( f_1 \) or \( f_2 \) is selected depending on both on the type of \( E \) or how \( \text{Id}(E) \) is used.
- Either \( S_1 \neq S_2 \) or \( T_1 \neq T_2 \) (or both).
- Ambiguity is not always resolved:

```ada
x : Float := (7/2)/(5/2);
```

Has at two ambiguous interpretations:

- \( 3/2 \approx 1.5 \)
- \( 3.5/2.5 \approx 1.4 \)

3.2.4 Summary

- Type system
- Recursively defined type systems
- Types define sets of values, but not all sets of values are types
- Purpose of type: Taxonomy, Legality, Semantics, Representation
- Type errors vs. pseudo type errors.
- Representation
  - Opaque
  - Transparent
  - Semi-transparent
- General overloading of terms vs. overloading of functions (identifiers) and operators.
- Resolution of overloading ambiguity
  - Ada context dependent
  - C++ context independent

References

- Data type
- Lisp machine
- Tagged architecture
- Type punning
- Type safety
- Type system
Exercises

1. What is the difference between value and type?
2. Is it true that $v \in \forall x \in T \Rightarrow \exists T \ni v \in T$? Explain.
3. What’s the difference between a “value” and a “variable”?
4. Does it follow from the fact that cardinality of the set of all PASCAL programs in $\mathbb{K}_0$, that $\# \mathbb{PASCAL} = \mathbb{K}_0$?
5. What are the three purposes of a type system?
6. Compare the terms “type error” and “type punning”.
7. For which $v$ it holds that $\text{types}(v) = \emptyset$? explain.
8. What’s the difference between a “value” and a “variable”?
9. “describe data effectively”, explain using examples if necessary.
10. Provide an example of a value $v$ for which $1 < \# \text{types}(v) < \infty$.
11. Enumerate all qualities that distinguish between a type and a “set of values”.
12. This C function does not produce type errors. Explain why not.

```c
short poem() {
    float home;
    return home;
    static dear;
    long ****time, no, cl;
    "Do␣you␣love␣me"? "Yes": "or␣no";
    "I␣love␣you";
    "I␣love␣you", 2;
    volatile short question = * "How␣much";
    1000 < "times" > dear;
    return question;
}
```

(Give one or two examples how the program uses overloading to make function `poem` read like a poem; is this type overloading or something else?)

3.3 The type constructors of Mock

Mock is not a real PL, so we can go wild:
- idealization.
- the “abstract” notion behind each type constructor
- loop holes in many of the definitions
- model for achievement

In many ways, ML is a practical version of Mock.

3.3.1 Power sets

**Definition 3.3.1** (Power set type constructor). If $T \in T$ is a type, then so is $\wp(T)$, its power set comprising all subsets of $T$, i.e.,

$$\wp(T) = \{ T' \mid T' \subseteq T \}.$$  \hfill (3.3.1)

An alternative notation

$$\wp(T) = 2^T$$ \hfill (3.3.2)

We have,

$$\#(\wp(T)) = 2^\#T$$ \hfill (3.3.3)

where $\#T$ denotes the cardinality of $T$.

PASCAL is probably the only language which natively supports this type constructor.

**71. Definition of type constructors is not enough**

Types worth nothing without operators/functions
- for creating values
- for manipulating values

**72. Value constructors for power sets**

Value constructors are operators which take values of $T$ and return values of type $\wp(T)$
- Nullary value constructor. The empty set, $\emptyset$, is a value of all power sets:
  $$\forall T : \emptyset \in \wp(T)$$ \hfill (3.3.4)
  in fact, $\emptyset$ is a literal.
- Unary value constructor. Let’s use the curly brackets:
  $$\forall v \in T : \{v\} \in \wp(T)$$ \hfill (3.3.5)
- n-ary value constructor. Generalizes the unary value constructor:
  $$\forall v_1, v_2, \ldots, v_n \in T, n \geq 1 : \{v_1, v_2, \ldots, v_n\} \in \wp(T)$$ \hfill (3.3.6)

**73. Defining operators on values of power sets**

Let $v \in T$, and and $u, u_1, u_2 \in \wp(T)$ be arbitrary. Then,
- Testing for membership: $v \in u$
- Testing for set equality: $u_1 = u_2$
- Set union operator: $u_1 \cup u_2 \in \wp(T)$
- Set intersection operator: $u_1 \cap u_2 \in \wp(T)$
- Set difference operator: $u_1 \setminus u_2 \in \wp(T)$
- Set complement operator: $\cap \in \wp(T)$
• Are power sets really useful?
• Can they be implemented efficiently?
• How should programmers use their plain keyboards to key in code phrases such as “∅”, “v ∈ u”, and “π”?

Real PLs must deal with these issues.

3.3.2 Cartesian product

7 Frames: Taxonomy of the type constructors of Mock Operators for the type constructors of Mock

Definition 3.3.2 (Cartesian product type constructor). If $T_1$ and $T_2$ are types, their Cartesian product is a type denoted by $T_1 \times T_2$; values of $T_1 \times T_2$ are

$$T_1 \times T_2 = \{ \langle v_1, v_2 \rangle | v_1 \in T_1; v_2 \in T_2 \}. \quad (3.3.7)$$

Note that

$$\#(T_1 \times T_2) = (\#T_1) \times (\#T_2) \quad (3.3.8)$$

Composing Given $v_1 \in T_1$, $v_2 \in T_2$ the binary composition operator $\langle - , - \rangle$ creates a value $\langle v_1, v_2 \rangle \in T_1 \times T_2$, i.e.,

$$v_1 \in T_1, v_2 \in T_2 \Rightarrow \langle v_1, v_2 \rangle \in T_1 \times T_2 \quad (3.3.9)$$

Decomposing two unary decomposition operators

- $(\cdot)\#1$
- $(\cdot)\#2$

If $v = \langle v_1, v_2 \rangle \in T_1 \times T_2$, then

- $v\#1 = v_1$
- $v\#2 = v_2$

Thus, we have

$$v \in T_1 \times T_2 \Rightarrow v\#1 \in T_1 \land v\#2 \in T_2 \quad (3.3.10)$$

76. Operators for product type

77. Cartesian products of three or more types

The Cartesian product type constructor is easily generalized to more than two types.

Commutativity? Never!

Associativity? Depending on the PL semantics:

- Structural $R \times S \times T = R \times (S \times T) = (R \times S) \times T$
- Nominal $R \times S \times T \neq R \times (S \times T) \neq (R \times S) \times T \neq R \times S \times T$

Nominal semantics is more common.

3.3.3 Integral exponentiation

50 Frames: Tuples vs. records in ML Integral exponentiation gives rise to array values Array values in Pascal Array values C Does C’s typedef brand types? Does Pascal’s TYPE definition brand types? Pascal offers automatic coercion from and to branded type Using C’s structures for branding The hotel thermometer Temperature tagging: why unions must be disjoint? The 3 operations on disjoint union in ML Disjoint union in Pascal Tagging in choice types Missing tag in Pascal variant record Tagging in ML vs. C (and Pascal) Unsafety of C’s union And the output is... Tags in C Choice & enumerated types Quiz: why wouldn’t this work in C++ Choice type in the implementation of Lisp Lisp’s original implementation Types for m-Lisp in C Testing these type definitions The “Cons” pool Types for m-Lisp in C: Cons records Types for m-Lisp in C: string handles Representing pointers with choice type Operations on pointers Void safety Syntax of variant records in Pascal Unsafety of Pascal’s variant record Inherent unsafety with choice types Safe decomposition of a variant record Choice types in some languages Emulating Unit type in C Type Unit in contemporary languages Incomplete types Using void in C & other subtle points Why void is not a C (or JAVA) type? Bottom: examples of use? Emulating None in C Types None & Any in Eiffel Eiffel inheritance clause Type Any in C++ Mappings: arrays Cardinality of mapping Mappings: multidimensional arrays & currying Mappings: single argument function Mappings: many arguments functions Programmatic functions vs. mathematical functions Representation of power set values Type of Prolog values in ML ML types for vProlog: system of equations Simplified version Type of Lisp values in ML Lists: the classical recursive data type Recursive definitions of lists Recursive definition of trees

78. Equality of types?

Even if $x$ and $y$ are “essentially” the same, a fully formal definition may force the claim $x \neq y$.

From a practical point of view, the following two types are equivalent:

- $T_1 \times T_2 \times T_3$
- $(T_1 \times T_2) \times T_3$

Can we write $T_1 \times T_2 \times T_3 = (T_1 \times T_2) \times T_3$?

• PL tend to be idiosyncratic in their definition of equality.
• It takes non-trivial language design effort to make the two types equal.
• Many languages don’t bother.

79. Integral exponentiation

Integral exponentiation makes homogeneous tuples: a Cartesian product where all the tuple components are chosen from the same type.

Definition 3.3.3 (Integral exponentiation type constructor). For a type $T \in \mathbb{T}$ and $n \in \mathbb{N}$, the integral exponentiation of $T$ to the power of $n$, $T^n$, is defined by

$$T^n = T \times \cdots \times T \quad (3.3.11)$$
Observe that
\[ \#(T^n) = (\#T)^n. \] (3.3.12)

### 80. Operators for integral exponentiation

**Composition** Given values \( v_1, v_2, \ldots, v_n \in T \), the composition operator \([v_1, v_2, \ldots, v_n] \) evaluates to a value \([v_1, v_2, \ldots, v_n] \in T^n \).

**Decomposition** Given a value \( v = [v_1, \ldots, v_n] \in T^n \) and an expression \( e \) of type integer, the \( v!e \) is \( v_i \), where \( i \) is the value to which \( e \) evaluates.

**Issues:** How should MOCK deal with

**Type error** \( e \) is not an integer.

**Pseudo type error** \( i < 1 \) or \( i > n \) (array index underflow/overflow).

**Missing operations** programmers need insertion, appending, merging, and many other operations to generate values of type \( T^n \).

### 3.3.4 Unit type

#### 81. Unit type

What does one mean by \( n \in \mathbb{N} \) in the definition of integral exponentiation? For a type \( T \in \mathbb{T} \) and

- How should MOCK define \( T^1 \)? Is \( T^1 = T? \)

**Oops!** Not a very interesting case; PLs make arbitrary decisions.

- What does \( T^0 \) mean? Is it the same for all \( T? \)

**Yes!** It is a useful and interesting type.

**Definition 3.3.4 (The Unit type).** The **Unit type** is \( T^0 \), where \( T \) is some type; alternatively, **Unit** is a Cartesian product of zero types.

**Unit** can be thought of as

**Composite type**

- Cartesian product of \( n \) types where \( n = 0 \)
- Exponential to the 0th power (where the “component” is arbitrary)

**Atomic type**

#### 82. Properties of Unit

- **Unit** is the neutral element of Cartesian product
- **Unit** is not the empty set, \( \text{Unit} \neq \emptyset \).

\[ \#\text{Unit} = 1 \] (3.3.13)

- Unit has exactly one value: the 0-tuple:
  \( \text{Unit} = \{()\} \). (3.3.14)

- A **Unit** “variable” is not really a variable, since it can store only one value, which can never be changed.

- \( \#\text{bits required to store a value of type Unit:} \)
  \( \lg_2 |\text{Unit}| = \lg_2 1 = 0. \) (3.3.15)

### 3.3.5 Branding

#### 83. Motivation for branding

It is often the case that we want to make a new type of an existing type, without adding anything new to the type definition:

The **MKSC system of Units**

**Length** A real number, designating *meters*

**Mass** A real number, designating *kilograms*

**Time** A real number, designating *seconds*

**Coulomb** A real number, designating *electrical charge*

#### 84. Branding type constructor

Let \( I \) be an infinite set of identifiers (often called *labels*).

**Definition 3.3.5 (Branding).** If \( T \in \mathbb{T} \) is a type and \( \ell \in I \) is label then \( \ell(T) \) is the \( \ell \) brand of \( T \) where,

\[ \ell(T) = \{ (\ell, v) \mid v \in T \}. \] (3.3.16)

**Characteristics:**

\[ \forall \ell \in I: T \neq \ell(T) \] (3.3.17)

\[ \forall \ell_1, \ell_2 \in I: \ell_1 \neq \ell_2 \iff \ell_1(T) \neq \ell_2(T) \] (3.3.18)

#### 85. Operators for branding

#### Creation

A value of type \( \ell(T) \) is created from a value \( v \in T \)

\[ v \in T \Rightarrow \ell(v) \in \ell(T) \] (3.3.19)

#### Extraction

A value \( v \in T \) can be extracted from type \( \ell(T) \)

\[ \ell(v) \in T \Rightarrow \ell(v) \#\ell \in T \] (3.3.20)

### 3.3.6 Records

**8 Frames:** Choice type in the implementation of Lisp. Lisp’s original implementation. Types for \( \mu\text{-Lisp} \) in C. Testing these type definitions. The “Cons” pool. Types for \( \mu\text{-Lisp} \) in C: Cons records. Types for \( \mu\text{-Lisp} \) in C: string handles. Representing pointers with choice type. Operations on pointers. Void safety. Syntax of variant records in Pascal. Unsaicety of Pascal’s variant record. Inherent unsafety with choice types. Safe decomposition of a variant record. Choice types in some languages.
86. Labeled Cartesian products: records

**Cartesian products** positional access to components: 1st component, 2nd component, 3rd component, 4th component, etc.

**Records** Access component by (hopefully, meaningful) name

**Definition 3.3.6 (Record type constructor).** Let 
\[ \ell_1, \ldots, \ell_n \in \mathbb{I}, \]
for \( n \geq 0 \) be unique labels. Let 
\[ T_1, \ldots, T_n \]
be types. Then, 
\[ \{\ell_1 : T_1, \ldots, \ell_n : T_n\} \]
is the record type induced by the labels \( \ell_1, \ldots, \ell_n \) and types \( T_1, \ldots, T_n \).

Record types can be thought of as product of brands:
\[ \{\ell_1 : T_1, \ldots, \ell_n : T_n\} = \ell_1(T_1) \times \cdots \times \ell_n(T_n) \] (3.3.21)

87. Operators for record type

Composition and decomposition operators are easy to define...

**Composition** values of the record type 
\[ \{\ell_1 : T_1, \ldots, \ell_n : T_n\} \]
are created by the \( n \)-ary operator \( \{\ell_1 : (\cdot), \ldots, \ell_n : (\cdot)\} : \)
\[ \forall v_1 \in T_1, \ldots, v_n \in T_n : \]
\[ \{\ell_1 : v_1, \ldots, \ell_n : v_n\} \in \{\ell_1 : T_1, \ldots, \ell_n : T_n\} \] (3.3.22)

**Decomposition** Given a value, \( \{\ell_1 : v_1, \ldots, \ell_n : v_n\} \), the unary decomposition operator \( (\cdot)\#\ell_i \) (for all \( 1 \leq i \leq n \)) evaluates to \( v_i \):
\[ \forall i, 1 \leq i \leq n : \{\ell_1 : v_1, \ldots, \ell_n : v_n\}\#\ell_i = v_i \] (3.3.23)

3.3.7 Disjoint union

**Definition 3.3.7 (Union type constructor).** If \( T_1, T_2 \in \mathbb{T} \) are types, then so is their union, \( T_1 \cup T_2 \).

- **Problem**: what if \( T_1 \) and \( T_2 \) are not disjoint?
- In particular, they may even be equal...
- Suppose that value \( v \in T_1 \), then \( v \) also belongs to \( T_1 \cup T_2 \), but how do we know whether it belongs to \( T_1 \) or to \( T_2 \)?

The union must be disjoint!

88. Union type constructor

89. Choice types: disjoint union type constructor

**Definition 3.3.8 (Choice type).** If \( T_1, T_2 \in \mathbb{T} \) are types, then so is their disjoint union, \( T_1 + T_2 \), defined by the set
\[ T_1 + T_2 = \ell_1(T_1) \cup \ell_2(T_2) \] (3.3.24)
and where \( \ell_1, \ell_2 \in \mathbb{I}, \ell_1 \neq \ell_2 \) are some labels.

The notation follows from the fact that
\[ \#(T_1 + T_2) = \#T_1 + \#T_2. \] (3.3.25)

Labels:
- often called tags in the context of disjoint unit
- used for telling whether \( v \in T_1 + T_2 \) came from \( T_1 \) or \( T_2 \).
- are arbitrary; any \( \ell_1, \ell_2 \) will do as long as \( \ell_1 \neq \ell_2 \)

90. Type equality hardship, again

**Definition 3.3.9 (Enumerated type constructor).** If \( \ell_1, \ell_2, \ldots, \ell_n \in \mathbb{I}, n \geq 1 \) are labels, then 
\[ \{\ell_1, \ell_2, \ldots, \ell_n\} \]
is an enumerated type, whose values are 
\( \ell_1, \ell_2, \ldots, \ell_n \).

An enumerated type can be thought of as a disjoint union of branded Unit types:
\[ \{\ell_1, \ell_2, \ldots, \ell_n\} = \ell_1(\text{Unit}) + \ell_2(\text{Unit}) + \cdots + \ell_n(\text{Unit}) \] (3.3.26)

91. Operators for choice type

Let \( T_1, T_2 \) be types

**Creation** Use the \( \ell_1(\cdot) \) and \( \ell_2(\cdot) \) operators
\[ v_1 \in T_1 \Rightarrow \ell_1(v_1) \in T_1 + T_2 \]
\[ v_2 \in T_2 \Rightarrow \ell_2(v_2) \in T_1 + T_2 \] (3.3.27)

**Checking** Use the \( ?\ell_1 \) and \( ?\ell_2 \) unary operators
\[ v \in T_1 + T_2 \Rightarrow ?\ell_1 \begin{cases} \text{true} & v = \ell_1(v_1), v_1 \in T_1 \\ \text{false} & v = \ell_2(v_2), v_2 \in T_2 \end{cases} \] (3.3.28)

Operator \( ?\ell_2 \) is defined similarly

**Decomposing** Given \( v = \ell_1(v_1) \in T_1 + T_2 \), the unary decomposition operator \( \#\ell_1 \) evaluates to \( v_1 \in T_1 \).
Operator \( \#\ell_2 \) is defined similarly

**Type error** when evaluating \( (v)\#\ell_1 \) in the case that \( v = \ell_2(v_2), v_2 \in T_2 \)
3.3.8 Type None and type Any

**Definition 3.3.10** (The Bottom type). Type None, also known as Bottom, and often denoted ⊥, is the empty set, i.e.,

\[
\text{None} \equiv \text{Bottom} \equiv \bot \equiv \emptyset.
\]

Obviously,

\[
\#\bot = 0, \quad \text{(3.3.29)}
\]

Type \(\bot\) is...

• derived from the choice constructor just as Unit is derived from Cartesian product
  
  ... the neutral element of the choice type constructor

• Cardinality is zero!
  
  ... no legal values

93. The Any type

**Definition 3.3.11** (The Any type). Type Any, also known as All, or Top, and often denoted \(\top\), is the universal set, i.e., for a language \(L\) with values’ universe \(\mathbb{V}_L\),

\[
\text{Any} \equiv \top = \bigvee L.
\]

Type \(\top\): The type of any arbitrary value that \(L\) may generate

\[
\begin{align*}
\forall T \in \top: T \times T &\subseteq \top \\
\forall T \in \top: \bot \times T &= \bot \\
\forall T \in \top: T + T &= \top \\
\forall T \in \top: \bot + T &= T
\end{align*}
\]

3.3.9 Mapping types

94. Mappings vs. partial-mapping

**Definition 3.3.12** (Mapping). A mapping (also called function) from a set \(S\) to a set \(T\) is a set \(m \subseteq S \times T\), that associates precisely one value of \(T\) with each value of \(S\), i.e.,

\[
\forall s \in S: \{t \mid (s, t) \in m\} = 1
\]

**Definition 3.3.13** (Partial mapping). We say that the set \(m \subseteq S \times T\) is a partial mapping (partial function), from set \(S\) to set \(T\) if it never associates more than value of \(T\) with any value of \(S\), i.e.,

\[
\forall s \in S: \{t \mid (s, t) \in m\} \leq 1
\]

If \(m\) is a partial mapping, there there may be members of \(S\) for which \(m\) associates no members of \(T\).

95. Mapping & partial mapping

**Definition 3.3.14** (Mapping (partial mapping) type constructor). Let \(T\) and \(S\) be types. Then, (partial) mapping from \(S\) to \(T\), denoted \(S \to T\) (sometimes also \(T^S\)) is

\[
S \to T = \{m \mid m\ \text{is a (partial) mapping from} \ S \ \text{to} \ T\}.
\]

(3.3.36)

Ideally,

\[
\not\exists T = \top \to \text{Boolean}
\]

96. Mapping is similar to exponentiation

We have,

\[
\#(S \to T) = \#T^{\#S}.
\]

(3.3.38)

Suppose that we write the

\[
S \to T = T^S
\]

(3.3.39)

Then, currying

\[
(S_1 \times S_2) \to T = S_1 \to (S_2 \to T)
\]

looks very much like the exponential identity

\[
T^{S_1 \cdot S_2} = (T^{S_2})^{S_1}.
\]

(3.3.40)

(3.3.41)

97. Mapping and integral exponentiation

Let \(n\) denote the type obtained by taking the \(n\) sized subrange of the integer type

\[
n = 1, \ldots, n.
\]

(3.3.42)

Then, the mapping \(\mathbb{N} \to \mathbb{R}\) which is the type of a real array in Fortran\(^{11}\) is isomorphic to \(\mathbb{R}^n\), i.e.,

\[
n \to \mathbb{R} = \mathbb{R}^n
\]

(3.3.43)

\(^{11}\)Unlike C, the first index of a Fortran array is 1
Ain’t it fortunate that we allow ourselves to write the mapping \( n \rightarrow \mathbb{R} \) also as \( \mathbb{R}^n \)?

Note that

\[
\text{Unit} = 1. \quad (3.3.44)
\]

and,

\[
\text{None} = 0. \quad (3.3.45)
\]

98. Unifying equation

Euler’s equation involving the five most important constants of mathematics

\[
e^{ix} + 1 = 0. \quad (3.3.46)
\]

The type theory equivalent

\[
\tau^1 = 1. \quad (3.3.47)
\]

There is precisely one function that maps the None type to the All type. (This function is empty, but should we care?)

3.3.10 Recursive type constructor

Definition 3.3.15 (Recursive type definition).

- Let \( T_1, \ldots, T_m \) be some fixed types.
- Let \( \tau_1, \ldots, \tau_n \) be type unknowns.
- Let \( E_1, \ldots, E_n \) be type expressions involving types \( T_1, \ldots, T_m \) and unknowns \( \tau_1, \ldots, \tau_n \).
- Then, the system of equations

  \[
  \tau_1 = E_1(T_1, \ldots, T_m, \tau_1, \ldots, \tau_n)
  \]

  \[
  \vdots
  \]

  \[
  \tau_n = E_n(T_1, \ldots, T_m, \tau_1, \ldots, \tau_n)
  \]

defines new types \( \tau_1, \ldots, \tau_n \) as the “minimal” solution of this system.

100. Specifics of the definition of the “recursive type constructor”

What is allowed in the “type expressions” \( E_1, \ldots, E_n \)?

- Cartesian product
- Disjoint union
- Mapping
- : ...
- Not all expressions are allowed
- We will not deal with the specifics here.

Kinds of recursive type constructor:

Simple \( n = 1 \), create a single type

General \( n \geq 1 \), create multiple types

Consider the case \( n = 1 \), i.e., an equation with only one type variable, \( \sigma \)

\[
\sigma = E(T_1, \ldots, T_m, \sigma)
\]

then, we build the following approximations,

\[
\sigma_0 = \emptyset = \bot
\]

\[
\sigma_1 = E(T_1, \ldots, T_m, \sigma_0) = E(T_1, \ldots, T_m, \bot)
\]

\[
\sigma_2 = E(T_1, \ldots, T_m, \sigma_1) = E(T_1, \ldots, T_m, E(T_1, \ldots, T_m, \bot))
\]

\[\vdots\]

\[
\sigma_{n+1} = E(T_1, \ldots, T_m, \sigma_n) = E(T_1, \ldots, T_m, E(T_1, \ldots, T_m, \sigma_{n-1}))
\]

\[\vdots\]

\[
\sigma = \bigcup_{i=0}^{\infty} \sigma_i.
\]

102. Solving the list recursive equation

For brevity,

1. let \( \tau \) denote the unknown,
2. write \( \mathbb{Z} \) instead of \( \text{int} \)
3. write \( \mathbb{M} \) instead of \( \text{nil} \)
4. write \( \zeta \) instead of \( \text{cons} \)

\[
\tau = \mu + \zeta(\mathbb{Z} \times \tau)
\]

\[
= \mu + \zeta(\mathbb{Z} \times (\mu + \zeta(\mathbb{Z} \times \tau)))
\]

\[
= \mu + \zeta(\mathbb{Z} \times \mu) + \zeta(\mathbb{Z} \times \zeta(\mathbb{Z} \times \tau))
\]

\[
= \mu + \zeta(\mathbb{Z} \times \mu) + \zeta(\mathbb{Z} \times \zeta(\mathbb{Z} \times \mu)) + \cdots
\]

103. Does it converge?

Many (boring) theorems and (even more boring) proofs regarding

- “convergence”
- uniqueness of solution
- independence in the order of application
- proper structure of \( E_1, \ldots, E_n \)

Not discussed here
104. Taylor series “solution” of recursive type equations

\[ t = (1 + T) = 1 + Z * t * t \]  \hspace{1cm} (3.3.51)

Figure 3.3.1: Using Mathematica to solve the quadratic equation \( \tau = 1 + Z \tau^2 \) for \( \tau \), the recursive binary tree type

\[ \tau = 1 + Z + 2Z^2 + 5Z^3 + 14Z^4 + 42Z^5 + \cdots \]  \hspace{1cm} (3.3.52)

3.3.11 Summary

105. Understanding the Taylor series expansion

106. Taxonomy of the type constructors of Mock

![Figure 3.3.2: Distinct topologies of binary trees with with \( n \) nodes; \( n = 0,1,2,3 \); each node stores an integer \( i \in \mathbb{Z} \)]

![Figure 3.3.3: Taxonomy of the type constructors of Mock]

**Theory vs. practice**

Type constructors in real PLs are not so elegant; their shape is invariably a compromise between many conflicting practical demands.

<table>
<thead>
<tr>
<th>Type constructor</th>
<th>Notation</th>
<th>Value composition</th>
<th>Value decomposition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power set</td>
<td>( \wp T )</td>
<td>( {\ldots} )</td>
<td>#1, #2, \ldots, #n</td>
</tr>
<tr>
<td>Product</td>
<td>( T_1 \times T_2 \times \cdots \times T_n )</td>
<td>( \langle \ldots \rangle )</td>
<td>#1, #2, \ldots, #n</td>
</tr>
<tr>
<td>Integral</td>
<td>( T^* )</td>
<td>( [\ldots] )</td>
<td>#i</td>
</tr>
<tr>
<td>Exponentiation</td>
<td>( f(T) )</td>
<td>( \ell(\cdot) )</td>
<td>#i</td>
</tr>
<tr>
<td>Branding</td>
<td>( (T_1, \ldots, T_n) )</td>
<td>( \ell_1(\cdot), \ldots, \ell_n(\cdot) )</td>
<td>#1, #2, \ldots, #n</td>
</tr>
<tr>
<td>Records</td>
<td>( (T_1, T_2, \ldots, T_n) )</td>
<td>( (\ldots) )</td>
<td>#i</td>
</tr>
<tr>
<td>Disjoint</td>
<td>( T_1 + T_2 + \cdots + T_n )</td>
<td>( \ell(T_1) \cup \cdots \cup \ell(T_n) )</td>
<td>#i, #1, #2, \ldots, #n</td>
</tr>
<tr>
<td>Union</td>
<td>( T_1 \cup T_2 \cup \cdots \cup T_n )</td>
<td>( \ell(T_1) \cup \cdots \cup \ell(T_n) )</td>
<td>#i</td>
</tr>
<tr>
<td>Mapping</td>
<td>( S \rightarrow T )</td>
<td>( S \rightarrow T )</td>
<td></td>
</tr>
<tr>
<td>Subrange</td>
<td>#i</td>
<td>#i</td>
<td></td>
</tr>
<tr>
<td>Simple</td>
<td>( \tau = E(\tau_1, \ldots, \tau_n) )</td>
<td>( \tau_1 = E_1(\tau_1, \ldots, \tau_n) )</td>
<td></td>
</tr>
<tr>
<td>Recursive</td>
<td>( \tau = E(\tau_1, \ldots, \tau_n) )</td>
<td>( \tau_1 = E_1(\tau_1, \ldots, \tau_n) )</td>
<td></td>
</tr>
<tr>
<td>Multiple</td>
<td>( \tau = E(\tau_1, \ldots, \tau_n) )</td>
<td>( \tau_1 = E_1(\tau_1, \ldots, \tau_n) )</td>
<td></td>
</tr>
</tbody>
</table>

Table 3.3.1: Taxonomy of the type constructors of MOCK
References

• Algebraic types
• Bottom type
• Currying
• Disjoint union
• Empty type
• Function type
• Nested function
• Recursive data type
• Set type constructor
• Tagged union
• Top type
• Tuple types
• Type system
• Unit type
• Void safety
• Void type

Exercises

1. What’s the type of the composition operator for tuple type? and, for record type?
2. Define a set of operators for decomposing power sets in MOCK.
3. Define a set of operators for mappings in MOCK.
4. Why is that the disjoint union type constructor does not have a positional equivalent, in the same manner that the record type constructor has a positional equivalent in the form of tuple type constructor?
5. Explain why it makes sense that PLs for matrix, vector and other mathematical computation would not distinguish between a value of an array of of length 1 and a scalar value.
6. Can you explain why arrays with a variety of index types (as in PASCAL and ADA) seem to have vanished?
7. Recall the notion of type errors: the expression \( v\#3 \) is a type error when \( v \in \ell(T) \). Which type errors can mappings produce?
8. Why doesn’t EIFFEL have a “Choice” type?
9. Provide an example\(^{12}\) for a composite type whose values are atomic?
10. Provide an example\(^{13}\) for a atomic type whose values are composite?
11. Provide an example\(^{14}\) for a composite type whose values are composite? Can you tell whether the values are composed in the same way that the type is composed?
12. Why JAVA has no “Choice” type?
13. Explain why languages designed for string processing would not distinguish between an array of length 1 of characters and a scalar value.
14. Explain how it is possible to view functions with more than one parameter in, e.g., PASCAL, as taking a tuple type argument. Refer both to the definition of the function and its invocation.
15. What is the difference between type aliasing and type branding?
16. Explain why many examples in this unit mention variables, despite the fact that this course did not discuss variables yet?

3.4 Type constructors in actual PLs

7 Frames: □ Case study: how many characters are there? □ Case study: The ancient IBM 704 BCD character encoding □ Evolution of character sets □ Is Unicode the answer? □ Challenges in PL design □ Impact of character encoding on PLs I □ Impact of character encoding on PLs II

3.4.1 Product type constructor

ML tuples

\[
\text{type person = string * string * int * real}
\]

\[
\text{if (#3 someone) >= 18 then ... else ...}
\]

\[
\text{val (surname, forename, age, height) = someone}
\]

\[
\text{if age >= 18 then ... else ...}
\]

ML records

\[
\text{type person = { surname: string, forename: string, age: int, height: real }}
\]

3.4.2 Integral exponentiation

Integral exponentiation is useful for understanding arrays. However,

- Integral exponentiation is meaningful only for languages like C and FORTRAN in which the indices or array are integers in the range 0,1,... (in C) or 1,2,... (in FORTRAN).
- In PASCAL any discrete type may serve as array index so integral exponentiation does not apply.

\[^{12}\text{i.e., a specific type in a specific PL and perhaps even values}\]

\[^{13}\text{ditto}\]

\[^{14}\text{ditto}\]
• Pascal, Fortran and C offer array variables
• We are still in the context of values, not variables.

110. Array values in Pascal

• Many PLs offer array variables;
• Array values are not always first class, e.g., in Pascal:
  – cannot create an array value, without the help of a variable.
  – cannot name an array value in the const section.
  – can pass array values as parameters to functions and procedures
  – but not return these
    * in original version, functions returned value via a designated register
    * arrays do not fit within a register
    * (records also do not fit, hence, similar limitation)
    * limitation removed in modern versions of Pascal.

111. Array values C

C also limits array values
• Array values initializer (can be thought of as array literal)
• no other array literals
• cannot pass array values as parameters to function
• cannot return arrays
• ...

3.4.3 Branding

112. Does C’s typedef brand types?

Definitions:

```c
typedef double Meters;
typedef double Seconds;
typedef double Kgs;
typedef double Coulombs;
```

Trying this out:

```c
VAR
  Meters m;
  Seconds s;
  Kgs k;
  Coulombs c;
PROCEDURE PRINT_SECONDS(s: Seconds);
Begin
  Write(s, "sec");
end;
BEGIN
  PRINT_SECONDS(0.3); // ✅
  PRINT_SECONDS(32); // ❌
END.
```

Conclusion: C’s TYPE definitions do not brand!

113. Does Pascal’s TYPE definition brand types?

Definitions:

```pascal
TYPE
  Meters = Real;
  Seconds = Real;
  Kgs = Real;
  Coulombs = Real;
```

Trying this out:

```pascal
VAR
  m: Meters;
  s: Seconds;
  k: Kgs;
  c: Coulombs;
PROCEDURE PRINT_SECONDS(s: Seconds);
Begin
  Write(s, 'sec');
  Write('
');
End;
BEGIN
  PRINT_SECONDS(0.3); // ✅
  PRINT_SECONDS(32); // ✅
END.
```

Conclusion: Pascal’s TYPE definitions do brand!

114. Pascal offers automatic coercion from and to branded type

Definitions:

```pascal
TYPE
  Seconds = Real;
PROCEDURE PRINT_SECONDS(s: Seconds);
Begin
  Write(s, "sec");
End;
BEGIN
  PRINT_SECONDS(0.3); // ✅
END.
```

Conclusion: Pascal’s TYPE definitions do brand!

115. Using C’s structures for branding

C’s typedef
• does not create a new type
• provides an alias for an existing type.

Use struct to create new types:

```c
struct Address { char *s; };
struct Name { char *s; };
struct Address a;
struct Name n;
a = n; // X
```

Conclusion: C’s typedefs do not brand!

3.4.4 Union and choice/disjoint union type constructors

116. The hotel thermometer

yet another tagging example Guests to Hotel California come from:

USA Use Fahrenheit
Israel Use Celsius
The room data structure

```c
struct Room {
    // Many room facilities
    struct Displays {
        // Many indicators
        union Temperature {
            float fahrenheit;
            float celsius;
        } temperature;
    } controls;
};
```

117. Temperature tagging: why unions must be disjoint?

- Temperature is a real number, either way
- Tagging is needed for safe decomposition.
- Simple set union: $\mathbb{R} \cup \mathbb{R} = \mathbb{R}$
- With tagging we obtain

$$\text{Temperature} = (\{\text{Celsius}\} \times \mathbb{R}) \cup (\{\text{Fahrenheit}\} \times \mathbb{R})$$

118. The 3 operations on disjoint union in ML

**Definition**

```c
datatype number = exact of int | approx of real;
```

**Value construction**

```c
exact(i + 1)
approx(r/3.0)
```

**Decomposing a construction**

```c
case n of
    exact i => i
    approx r => round(r);
```

119. Disjoint union in Pascal

In the PASCAL lingo “variant records”:

**A number type in PASCAL**

```pascal
type
Accuracy = (exact, approx);
Number = record case tag: Accuracy of
    exact: (ival: Integer);
    approx: ( rval: Real);
end;
```

Values of type **Number**:

$$\{\ldots, (\text{exact}, -2), (\text{exact}, -1), (\text{exact}, 0),
(\text{exact}, 1), (\text{exact}, 2), \ldots,
(\text{approx}, -1.0), \ldots, (\text{approx}, 0.0),
(\text{approx}, 1.0), \ldots\}$$

3.4.5 Tags in concrete PLs

120. Tagging in choice types

**Definition 3.4.1 (Tag).** Tag is the mechanism for storing the selection made in a choice type along with the value associated with the choice.

- Tagging of pointers ($T + \text{Unit}$) is implicit in all PLs, however, compiler enforced safety is rare.
- Tagging in PASCAL:

  **Original version** Compiler forcing a definition of a tag field.

  **Standard version** Tag field is optional (syntax reminds you of its necessity).

  **All versions** Compiler does not enforce safety.

121. Missing tag in Pascal variant record

**Figure 3.4.1:** Pascal variant record providing two perspectives of the same memory cells

**Reckless programmer assumptions**

- All cases in a variant record use the same memory address
- There is no alignment
- There is no padding
- Machine representation is in the order of definition

122. Tagging in ML vs. C (& Pascal)

**ML** Tagging is built into the language.

- **Tag is Implicit:** Cannot access a “Tag Field” directly.
- **Correct tagging Enforced:** no way to store a value into one choice selection and read it from another.

**C** Responsibility lies with programmer:

- **Definition:** define (or not) a tag field
- **Usage:** use (or don’t use) the tag field
- **Safety**: safe (or unsafe use) the tag field
  
i.e., the programmer is free to decide

- whether a tag field is defined, and if it is defined,
  - whether it is used or not, and if it is used,
  - whether it is used in a safe manner or not.

123. **Unsafety of C’s union**

```c
union {
    long int i;
    double d;
    unsigned char chars[sizeof(double)];
} u; // All fields occupy the same storage
#include <stdio.h>
main() {
    int i;
    u.d = 3.14159264590;
    printf("u.i=%ld\n", u.i);
    printf("u.d=%f\n", u.d);
    for (i = 0; i < sizeof(double); i++)
        printf("u.chars[%d] = %d = %c\n",
            i, u.chars[i], u.chars[i]);
}
```

124. **And the output is...**

Who knows?

On my machine, I got...

u.i=4614256656534729973
u.d=3.141593

This is what’s called digging deeply into the machine representation!

125. **Tags in C**

- There is no way of defining the tag in the `union` itself.
- Must wrap the tag in a `struct`
- We can see that `union` in C, is the nasty “union” type constructor, not the disjoint union we are after.

126. **Choice & enumerated types**

In ML, an enumerated type can be simulated as a choice between Units:

```
datatype suit =
    diamond of unit
  | heart of unit
  | spade of unit

datatype suit =
    diamond
  | heart
  | spade
  | clover;
```

127. **Quiz: why wouldn’t this work in C++**

```c
typedef union {
    struct {} diamond;
    struct {} heart;
    struct {} spade;
    struct {} clover;
} Suit;
```

128. **Choice type in the implementation of Lisp**

Why do we need set union types?

**Example: C’s representation of Lisp’s CONS Records**

```
struct PointerToAtomOrCons; // Forward declaration
typedef ... Atom; // Some definition of the ATOM type
struct Cons {
    PointerToAtomOrCons car;
    PointerToAtomOrCons cdr;
};
struct PointerToAtomOrCons {
    Huh??? // This should be either:
    // 1. Pointer to a struct Cons, or
    // 2. Pointer to an Atom,
    // but not both!
};
```

129. **Lisp’s original implementation**

- A CONS was a 36 bits machine word.
- The word was divided into equal size parts: CAR and CDR.
- CAR/CDR was further divided into two parts
  - 15 bits of pointer to “something”, which could be either a CONS word, or another ATOM.
  - 3 bits, telling which pointer it was.
### 130. Types for \( \mu \)-Lisp in C

Please, please, do not try to understand this in full now…

```c
// A more civilized way to name integer values:
enum {
    // How many bits for index into pool:
    LG2_POOLSIZE = 14,
    // How many bits for storing car/cdr kind:
    KIND_SIZE = 2
};
// Will be used for atoms:
char atoms[1 << LG2_POOLSIZE];
enum kind { NIL, STRING, INTEGER, CONS};
struct Cons {
    enum kind carKind: KIND_SIZE;
    unsigned int car: LG2_POOLSIZE;
    enum kind cdrKind: KIND_SIZE;
    unsigned int cdr: LG2_POOLSIZE;
} // Pool of struct Cons nodes:
pool[1 << LG2_POOLSIZE];
```

### 131. Testing these type definitions

```c
#include <assert.h>
#include <stdio.h>
...
main() {
    printf("A Cons record requires %ld bytes.\n", sizeof (struct Cons));
    printf("Our store requires %ld bytes.\n", sizeof pool);
    printf("It may contain up to %ld Cons entries.\n", sizeof pool / sizeof pool[0]);
    assert(1 << LG2_POOLSIZE ==
           sizeof pool / sizeof pool[0]);
    assert(4 ==
           sizeof (struct Cons));
}
```

### 132. The “Cons” pool

```c
enum { _ LG2_POOLSIZE = 14 _ };
struct Cons { _ pool[1 << LG2_POOLSIZE];
```

### 133. Types for \( \mu \)-Lisp in C: Cons records

```c
enum kind { NIL, STRING, INTEGER, CONS};
struct Cons {
    enum kind carKind: KIND_SIZE;
    unsigned int car: LG2_POOLSIZE;
    enum kind cdrKind: KIND_SIZE;
    unsigned int cdr: LG2_POOLSIZE;
} // Pool of `struct Cons` nodes:
pool[1 << LG2_POOLSIZE];
```

### 134. Types for \( \mu \)-Lisp in C: string handles

```c
enum { _ LG2_POOLSIZE = 14 _ };
const char *handles[1 << LG2_POOLSIZE];
```

### 135. Representing pointers with choice type

A pointer to type \( T \), either
- points to a value of type \( T \), or
- has a special value denoting that the pointer points nowhere

“Special” value of pointers?

- C 0
- Pascal `nil`
- Eiffel `void`
- Java `null`
- C++ `nullptr`

Pointers can be thought of as a choice between `Unit` type and \( T \)

### 136. Operations on pointers

**Definition 3.4.2 (Operations on pointers). Construction**

- Create a null pointer
- Create a pointer to a “stored” value

**Tag testing** Determine whether a pointer is null or not.

**Projection** If the pointer is not null, extract value.
137. Void safety

The property of a PL that a \texttt{null/nil/void} is never dereferenced.

<table>
<thead>
<tr>
<th>Java example</th>
</tr>
</thead>
</table>
| Grade g = null;  
- // Variable may or may not be initialized to a reference to a real object  
- g.factor(1.10); // May generate a runtime error |

**Java**  
Runtime check generating an error

**C++ pointers**  
No runtime check, an O/S error may be generated

**C++ references**  
Compile time guarantee

**C\# nullable types**  
Just like Java

**C\# non-nullable types**  
Just like C++ references

**Eiffel**  
Huge effort to ensure void safety

138. Syntax of variant records in Pascal

Syntax

```pascal
record case I: T of
  v1: (ℓ1: T1);
  ...
  vn: (ℓn: Tn);
end
```

where

- \( I \) is an identifier used for tagging
- \( T \) is its type (which must be “discrete”\(^{15}\))
- \( v_1, \ldots, v_n \) are values of type \( T \)
- \( ℓ_1, \ldots, ℓ_n \) are “field” names
- \( T_1, \ldots, T_n \) are their types

139. Unsafety of Pascal’s variant record

```
VAR n: Number;
Begin
  n.tag := exact; (* n.ival is still undefined *)
  (* n is now in an inconsistent state *)
  (* must not read n here *)
  ...
  n.ival := 7;
  (* n is now in a consistent state *)
  ...
  n.tag := approx; (* n’s value is changed in one step from (exact,7) to (approx,undefined) *)
  ...
  (* n is now in a consistent state *)
End
```

140. Inherent unsafety with choice types

With the absence of PL help, we may:

- Change the tag, without changing the value’s type.
- Change the value’s type, without changing the tag.
- Read a value as if it belongs a certain type, while the tag indicates otherwise.

These are a kind of type errors:

**Definition 3.4.3** (Type error). A program commits a type error if it makes an attempt to:

- manipulate \( v \) in a way which is inconsistent with \( T \).
- interpret machine representation of \( v \) in a way which is inconsistent with \( T \).

for some type \( T \) and a value \( v \in T \).

141. Safe decomposition of a variant record

Given a **Number**, return its **rounded** value

```
function roundNum(n: Number): Integer;
  case n.tag of
    exact: roundNum := n.ival;
    approx: roundNum := round(n.rval)
  end;
end;
```

142. Choice types in some languages

**C**  
**union** type constructor

**Pascal**  
**variant record**

**ML**  
**datatype** type constructor, e.g.,

```
datatype color = Red | Blue | Green;
datatype number = i of int | r of real;
```

datatype can be used for defining enumerated types as well.

**Java**  
Missing!

**Eiffel**  
Missing!

JAVA and EIFFEL lack union as typical to OO languages—the need for choice type constructors is diminished with inheritance.

3.4.7 Special types: **Unit**, **Top** & **Bottom**

143. Emulating **Unit** type in **C**

<table>
<thead>
<tr>
<th>Option I: singleton <strong>enum</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{typedef enum {unit} Unit;}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Option II: empty <strong>struct</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{typedef struct {} Unit;}</td>
</tr>
</tbody>
</table>

**This type’s (only) value is \{\}; it can only be used for initialization**

<table>
<thead>
<tr>
<th>Option III: zero sized array (only in C++)</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{typedef int Unit[0];}</td>
</tr>
</tbody>
</table>

\(^{15}\)e.g., Char, Integer, Boolean,...
144. Type **Unit** in contemporary languages

- Modern languages tend more and more let type **Unit** be a *first class* type
- this can make the language definition simpler,
- e.g., no distinction between functions and procedures:
  - **Unit** corresponds to the type **unit** of ML
  - **Unit** corresponds to the (incomplete) **void** type of C, denoted by its own keyword.

145. Incomplete types

- Incomplete types are types which do not provide information on their values.
- Type "**struct Person**" in the variable declaration
  
  ```
  struct Person *person;
  ```

  is *incomplete*

- An incomplete type can usually be completed, by specifying the missing information, but type **void** can never be completed!
- For historical reasons, `sizeof(void) = 1` when used in arithmetic of a pointer of **void** *

146. Using **void** in C & other subtle points

- **void** foo(int)
  - Is not a function that does not return anything
  - It is rather a function that can return in only one way, i.e., returns type **Unit**

- **int** bar(**void**)
  - Takes no arguments
  - Can be thought of as taking an argument of type **unit**

- **int** baz(int, ...);
  - function is declared to have a variable number of arguments

- **extern** boo()
  - return type is implicitly **int**
  - In K&R C, no declaration of arguments
  - In C++, function takes no argument (an argument of type **unit**)

147. Why **void** is not a C (or Java) type?

Yes, we have...

- **void** return
- **void** argument (but not in JAVA nor C++)

But,

- No **void** variables
- No **void** arrays
- No **void** fields in **struct**s.
- **void** * is not a pointer to a cell of type **void**.
- The language’s specification makes every possible attempt to avoid calling **void** a type.

In many ways, **void** is just a reserved word of C, which may look like a type.

148. Bottom: examples of use?

Actual PLs

- *meaningless as a variable’s type*
- C’s function `exit()`
  
  ```
  return type should be None rather than Unit or void.
  ```

More Examples?

- *that’s it! Only functions which never return are of type None*

Program analysis

- e.g., to prove that a variable may be uninitialized in a certain program location

149. Emulating **None** in C

Challenge: define a type with no legal values

**Option I: empty enum**

```
typedef enum {
    // empty!
} none;
```

**Option II: empty union**

```
typedef union {
    // empty!
} none;
```

The author of these slides is not so sure your C compiler will like these!

150. Types **None** & **Any** in Eiffel

- **All** types are classes
- Even **INTEGER** is a class
- **All** classes inherit from class **ANY**
- Class **NONE** inherits from all classes
- No class inherits from **NONE**
151. Eiffel inheritance clause

```eiffel
class ARRAYED_LIST [G] inherit
ARRAY[G]
export
(NONE) all ------ All features are inaccessible
(ANY) capacity ------ except for this feature
end
```

`ARRAYED_LIST[G]` inherits from `ARRAY[G]` while changing the `export` level of inherited features:
- All features are made private (exported to `NONE`)
- except `capacity` which is public (exported to `ANY`).

152. Type Any in C++

C++ offers minimal support for type `Any`:

- Function `printf` may take any number of arguments of any type, as long as the first argument is of type `const char *`.
- The notation `"..."` here refers to a list of any length (including zero), or arguments of type `Any`.

Catch all exceptions

```cpp
try {
    // do something
} catch (...) {
    printf("unfamiliar exception\n");
}
```

- The notation `"..."` here refers to an exception of type `Any`.
- Function `printf` will be invoked if the `try` block throws an exception of any type.

3.4.8 Mapping as functions and arrays

153. Mappings: arrays

```pascal
TYPE
Saying = (saysYes, saysMaybe, saysNo);
Meaning = (meansYes, meansMaybe, meansNo, identityCrisis);
Interpretation = Array[Saying] of Meaning;
```

With these types, a diplomat is defined by,

```pascal
VAR
diplomatInterpretation: Interpretation;
Begin
diplomatInterpretation[saysYes] := meansMaybe;
diplomatInterpretation[saysMaybe] := meansMaybe;
diplomatInterpretation[saysNo] := identityCrisis;
End
```

- Thus, variable `diplomatInterpretation` stores...
- (“Formally” we did not introduce the notion of `storage`, but it should be clear what it is.)
- Thus, variable `diplomatInterpretation` stores the value

```
(meansMaybe, meansMaybe, identityCrisis).
```

- This value is a tuple, a triple to be specific.
- Its type is `Meaning`³

Similarly, the representation of the interpretation of a professor sayings is

```
(meansNo, identityCrisis, meansYes).
```

Cardinality of mapping?

- So, we have a diplomat and a professor...
- How many possible different characters are there?
- How many different values are there of type `Interpretation`?
We have,

\[
\#\text{Saying} = 3 \\
\#\text{Meaning} = 4
\]

to specify an array of type \textit{Interpretation} you must provide value (one of four) to each tuple entry (three entries in total):

\[
\#\text{Interpretation} = 4^3 = \#\text{Meaning}^{\#\text{Saying}} = 64.
\]

155. Mappings: multidimensional arrays & currying

A multidimensional array type in Pascal

\[
\text{TYPE} \ M = \text{array}[S1, S2, S3] \text{ of } T;
\]

Type is

\[
S1 \times S2 \times S3 \to T,
\]

or with currying,

\[
S1 \to (S2 \to (S3 \to T)).
\]

By convention, the mapping operator is right associative:

\[
S1 \to (S2 \to (S3 \to T)) = S1 \to S2 \to S3 \to T. \quad (3.4.1)
\]

156. Mappings: single argument function

A PASCAL Function Definition

\[
\text{Function even}(n: \text{Integer}): \text{Boolean}; \quad \text{Begin} \\
\quad \text{even} := (n \text{ mod } 2) = 0 \\
\quad \text{end};
\]

Type is

\[
\text{Integer} \to \text{Boolean}.
\]

157. Mappings: many arguments functions

A recursive PASCAL function computing GCD

\[
\text{TYPE} \quad \text{Natural} = 1..\text{MAXINT}; \\
\text{Function GCD}(p, q: \text{Natural}): \text{Natural}; \quad \text{Begin} \\
\quad \text{If } p \text{ mod } q = 0 \text{ then} \\
\quad \quad \text{GCD} := q \\
\quad \text{else} \\
\quad \quad \text{GCD} := \text{GCD}(q, p \text{ mod } q); \\
\quad \text{end};
\]

Type is

\[
\text{Integer} \times \text{Integer} \to \text{Boolean}.
\]

158. Programmatic functions vs. mathematical functions

Functions in most PLs are an algorithmic implementations of mathematical mappings (also called \textit{functions}). However,

\textbf{Time} Programmatic functions may take time to compute

\textit{Runtime of Function GCD is } O(\log(p+q))

\textbf{Space} Programmatic functions are usually more memory efficient than the use of arrays for mapping.

\textit{An array implementation of GCD would require huge memory.}

\textbf{Side effects} Programming functions may have side effects

\textit{We can add WriteLn commands to Function GCD}

\textbf{Partial} Programmatic functions may not terminate

\textit{Are you absolutely sure that the Function GCD will terminate and not produce runtime error when presented with negative numbers?}

3.4.9 Power sets

159. Representation of power set values

If \#T = n,

- \#(\wp T) = 2^{\#T}.
- \(v \in \wp T\) requires \(n\) bits (with the simple \textit{bit-mask} representation)

\textbf{Set of Boolean} 2 bits

\textbf{Set of character} Ancient CDC 60 bits, which make one machine word.

\textbf{Modern Architectures} 256 bits (assuming ASCII) which make eight 32-bits words,

\textbf{with Unicode} \(\approx 100,000\) bits, which make an array of 3,000 integers

\textbf{Set of integer} Ancient CDC \(2^{60}\) bits

\textbf{Modern Architectures} \(2^{32}\) bits, which make \(2^{29}\) bytes, i.e., half a gigabyte.

3.4.10 Recursive types

160. Type of Prolog values in ML

Recall that in PROLOG, all values are \textit{terms}, where a term can be

- \textbf{composite} with
  - an \textit{atom} called functor, and
  - children terms
- an \textit{atom} (a string)
- a number which can real or integer
- a variable (a string)
**161. ML types for 

**Prolog: system of equations**

```ml
datatype Term = COMPOUND of (Atom * Terms)
             | ATOM of Atom
             | NUMBER of Number
             | VARIABLE of string
and
Terms = none
       | many of {first: Term , rest: Terms}
and
Number = INT of int
       | REAL of real
and
Atom = string;
```

**Characteristics**

Unknown types \((n = 4)\) Term, Terms, Number, and Atom

Known types \((m = 3)\) int, real, and string.

**162. Simplified version**

```ml
datatype Atom = string;
```

```ml
datatype Number = INT of int | REAL of real;
```

```ml
datatype Term = COMPOUND of (Atom * Terms)
             | ATOM of Atom
             | NUMBER of Number
             | VARIABLE of string
and
Terms = none
       | many of {first: Term , rest: Terms}
and
Number = INT of int
       | REAL of real
and
Atom = string;
```

**Characteristics**

Unknown types \((n = 2)\) Term and Terms

Known types \((m = 2)\) Atom, Number

**163. Type of Lisp values in ML**

In Lisp, all values are S-expressions, where an S-expression may be,

- **composite** with CAR and CDR which are S-expressions;
- an atom (a string)
- NIL

```ml
datatype SExpression = CONS of {CAR: SExpression , CDR: SExpression}
                      | ATOM of string
                      | NIL
```

**Characteristics**

Unknown types \((n = 1)\) SExpression

Known types \((m = 1)\) string

**164. Lists: the classical recursive data type**

```ml
datatype intlist = nil | cons of int * intlist
```

**Characteristics**

Unknown types \((n = 1)\) intlist

Known types \((m = 1)\) int

**Values of intlist:**

\[
\text{nil, cons}(11,\text{nil}) \\
\text{cons}(2,\text{cons}(3,\text{cons}(5,\text{cons}(7,\text{cons}(11,\text{nil}))))))
\]

These can be obtained by substituting values obtained so far in the right-hand side of the definition, to get new values.

**165. Recursive definitions of lists**

- Lists are very useful.
- Each list is finite, but there is no global limit on the number of elements (so there are unboundedly many lists defined here)
- ML has a predefined list type constructor. These are all valid ML types:

  ```ml
  int list
  bool list
  int list list
  ```

  - Operations include: test for emptiness, select head, select tail, concatenation, and length.

**166. Recursive definition of trees**

```ml
datatype T = leaf of int | branch of int*T*T
```

**Possible values:**

- leaf(11)
- branch(7,leaf(5),leaf(11))
- branch(7,leaf(5),branch(9,leaf(8),leaf(11)))

Set theoretical representation: type \(T\) is the minimal solution of the equation:

\[
\tau = \text{int} \cup (\text{int} \times \tau \times \tau) \quad (3.4.2)
\]
Exercises

1. Present a feature of `void` which incomplete types don’t have.

2. What does `nullptr` denote? Type, variable, value, or something else? Should it be considered an identifier?

3. Let $\mathcal{L}$ be a PL, then,
   - Enumerate all type constructors of $\mathcal{L}$
   - Which of these are modeled after the constructors mentioned in this subsection?
   - Which type constructors of $\mathcal{L}$ cannot be modeled after the constructors mentioned in this subsection?
   - Which of the constructors mentioned in this subsection, does not exist in $\mathcal{L}$? why?
   - Does $\mathcal{L}$ have a `Unit` type? is this type first class or not?
   - Does $\mathcal{L}$ have a `None` type? is this type first class or not?
   - Does $\mathcal{L}$ have an `All` type? is this type first class or not?

For

$\mathcal{L} = \text{C++}, \text{Pascal}, \text{ML}, \text{Go}, \text{MetaPost}, \text{D}, \ldots$

item Why is that very few languages have a power-set type constructor?

4. Does C++ offer a literal for the type `Unit`? If not, what would be a good notation for it?

5. Does ML offer literals for the type `Unit`? If not, what would be a good notation for it?

3.5 Atomic types

3.5.1 Taxonomy of atomic types

The type system is a set of subsets of the values’ universe

$$\mathcal{T} \subseteq \wp(\mathcal{V})$$

which is recursively defined:

**Atomic types** The building blocks; one cannot find any other type “within” an atomic type

**Type constructors** Create compound types from existing types atomic- and compound- types.

**Compound types** Types constructed from other types by employing type constructors;

A compound type must include “within” it:

- a construction rule
- another type, atomic or compound

(Except empty list of arguments to the construction rule)

Henceforth, we shall use the common term “primitive types” for builtin atomic types:

**Definition 3.5.1 (Primitive type).** A type which is both atomic and builtin, i.e., is not programmer defined, is called a primitive type.

**Note:** A PL could have builtin types which are not atomic, e.g., `string` in some PLs

---

**Builtin**
- e.g.,
  - `int` in C++
  - `integer` in Pascal

- AKA (Also Known As)
  - `primitive types`,
  - `basic types`,
  - `builtin types`
  - `rudimentary types`

**Programmer defined**
- mainly `enumerated types`
- found in Ada, Pascal, C,...
- Pascal example:

```pascal
TYPE Month = (January, February, March, April, May, Jun, July, August, September, October, November, December)
```

**Figure 3.5.1:** The term “primitive type” may be considered derogatory
Some “standard” (more or less) primitive types:

Character a character of some alphabet
String a sequence of characters
Boolean truth value
Integer an approximation of $\mathbb{Z}$
Natural an approximation of $\mathbb{N}$
Real an approximation of $\mathbb{R}$
Complex an approximation of $\mathbb{C}$

Fixed point A non-integral number with some fixed (decimal) accuracy, e.g., 2.75

Less “standard” atomic types
- Unit (also compound type: product of zero elements)
- None (also compound type: disjoint union of zero elements)
- Any (can be thought of as disjoint union of all possible types)

### 170. Classification of primitive types

In general, classification:
- is typically PL dependent
- helps the PL designer and PL user
  - communicate
  - remember PL rules
  - rationalize PL rules

### 171. Classification of primitive types

<table>
<thead>
<tr>
<th>Numeric</th>
<th>Non-numeric</th>
<th>Semi-numeric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetical operators: “*”, “+”, “-”,...</td>
<td>No arithmetical operators</td>
<td>Some arithmetical operators</td>
</tr>
<tr>
<td>int, real, complex</td>
<td>string, unit, none</td>
<td>date, time, pointer</td>
</tr>
</tbody>
</table>

Table 3.5.1: Classification of primitive types

Pointer, date, and time types are considered semi-numerical since they support (i) adding an integer (ii) subtracting an integer (iii) subtraction to find the difference between two dates, times, pointers

### 172. More classification: unordered, ordered & ordinal types

<table>
<thead>
<tr>
<th>ORDERED</th>
<th>UNORDERED</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comparison operators: “&lt;”, “&lt;=”, “&gt;”, “&gt;=”</td>
<td>No comparison operators</td>
</tr>
<tr>
<td>boolean, integer, real, date, time, character, string,...</td>
<td>complex, point</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ORDINAL</th>
<th>NONORDINAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Can be mapped to a subrange of $\mathbb{N}$</td>
<td>Cannot be mapped to a subrange of $\mathbb{N}$</td>
</tr>
<tr>
<td>boolean, int, pointer, string, character</td>
<td>date, time</td>
</tr>
</tbody>
</table>

Table 3.5.2: More classification: unordered, ordered & ordinal types

Ordinal types: (i) can serve as array indices; (ii) support “suc” and “pred” functions; (iii) support “++” and “--” operators; ...

### 173. “Ordinal” primitive types

- successor operator
- predecessor operator
- aka “discrete” types

Examples ASCII-Character, Integer, Natural,...
Non-Examples String, Unicode-Character, Real,...

Some languages offer more for ordinal types, e.g.,
- for loops in PASCAL
- switch in C and case...of in PASCAL
- array indices in PASCAL

are exclusive to ordinal types.

### 174. Testing for properties of a type

#### Trick: write a short program; does it compile?

**File BooleanIsOrdered.java**
```java
class BooleanIsOrdered {
    boolean foo() {
        return true <= false;  // X
    }
}
```

X The operator <= is undefined for the argument type(s)
boolean, boolean

**File CharIsNumerical.java**
```java
class CharIsNumerical {
    int foo() {
        return ':'-')'; // ✓
    }
}
```

compiles just fine!

**Traps:**
- Misunderstanding compiler error messages
- Idiosyncratic, non-standard obeying, compiler
- Special cases in the language definition
3.5.2 Set of primitive types as PL *i.d.*

- has a rich and expressive set of primitive types
- organizes these in a meaningful taxonomy
- Is ridiculous indeed!
- But why?

How should a language $\mathcal{L}$ select its set of primitive types?

**Programmer** Match intended use of $\mathcal{L}$

**Hardware** Two conflicting requirements:

- Efficiency allow $\mathcal{L}$’s programs to use the hardware efficiently
- Portability make $\mathcal{L}$’s programs portable

**Language** Design $\mathcal{L}$ to be coherent and easy to understand

176. The Mock PL...

The set of primitive types of $\mathcal{L}$ reflects

- the design principles
- the objectives
- the spirit

of $\mathcal{L}$.

177. Primitive types design: the “FEM” metaphor

**Ingredients:**

- **F** Flour
- **E** Eggs
- **M** Milk

Some combinations work great:

- Crêpe
- Blintzes
- Pancake

Yummy!! Others combinations are...

178. Tough questions of in the selection of primitive types

- How will programmers actually use $\mathcal{L}$?
- Which architectures will die?
- Which architectures will be born?
- Too many types? Language could be cumbersome
- Too few types? Programming may be awkward

The combination used in $\mathcal{L}$ are telling of $\mathcal{L}$’s

**Signature of PL design**

- The set of primitive types of $\mathcal{L}$ reflects
  - the design principles
  - the objectives
  - the spirit

  of $\mathcal{L}$.

179. Force I–Programmer: match intended use of $\mathcal{L}$

$\mathcal{L}$’s set of primitive types is telling its intended use

**C** Operating Systems/Systems Programming: Try to match hardware word size

**Fortran** Scientific computation: Choice of precision of real and complex numbers

**Cobol** Data processing: fixed length strings; fixed point numbers

**Snobol** String processing: Variable length strings.

**Excel** Spreadsheet: Text, Number, Date, Time, DATE-TIME, Currency, Logical (and no type constructors)

**Mock** ????

ASCII big bool complex128 complex256 complex64 date fixed10d2 fixed20d2 float16 float64 float80 int16 int32 int64 int8 message text time uint16 uint32 uint64 uint8 Unicode

Smörgåsbord

180. Case study: boolean & the intended programmer

**Background**

- Hardware does not\(^{18}\) have a Boolean type
- Programs rarely use data of type Boolean

  *type Boolean is used mainly for conditional commands*

**Approach**

**Pascal** educational purposes; good typed programming style; type Boolean is a must

\(^{18}\)booleans are realized by hardware as CPU “flags” which change after many operations and by instructions such as *jz* and *jge*

**Yuck!**
C no Boolean type; any atomic type is a Boolean in disguise

C++ Type bool introduced by community demand; attempt to “deprecate” automatic coercion of other types to bool

Java as in PASCAL

JVM (the Java Virtual Machine): as in C

3.5.3 Integral primitive types

Efficiency Make efficient use of the underlying hardware
- Match primitive types with those of the hardware
- Grant access to all hardware primitive types
- Intel is going to hate you unless you make efficient use of their hardware

Portability different architecture employ different types.

Issues:
- Varying word size:
  - 36, 48, 60,... in ancient architectures
  - 16, 32, 64, 128,... in more modern one
- Varying representation methods:
  - two’s complement
  - one’s complement

181. Force II–Hardware: efficiency vs. portability

182. Case study: “integer” in various PLs

1. Pascal: predefined Integer
- Very portable for toy programs
- In “serious” work, machine dependent value MAXINT gets in the way
- In one word, unportable

2. BCPL:
- Mother of all { } PLs
- Acronym: Basic Combined Programming Language
- Backronym: Before C Programming Language
- Only one primitive type, word
- A word is an integer, isn’t it?
- Matches whatever a machine word is

3. C/C++:
- Many integral types:
  - char
  - short
  - int
  - long
  - long long (on some dialects)
- Two Varieties:
  - signed
  - unsigned
- Hardware mapping must obey:
  - $(\text{char}) \leq (\text{short}) \leq (\text{int}) \leq (\text{long})$
  - $(\text{char}) = (\text{signed char}) = (\text{unsigned char})$
  - $(\text{short}) = (\text{unsigned short})$
  -...
- Type int is the most “natural” to hardware

supposedly, C makes the “right” portability/efficiency tradeoff
- Many many types
- Complex language rules
- Expression involving mixed types
- Converting each of the 8 types to another
- Bugs & confusion

4. Java: Similar to C/C++ but (tries to be) better
- Types:
  - byte (8-bit)
  - short (16-bit)
  - int (32-bit)
  - long (64-bit)
- No “unsigned” variant
- All types are two’s complement
- Fixed mapping to hardware
- Fixed “hardware”: the “Java Virtual Machine” (JVM)
- Many conversions, e.g., long to int
- Simplified conversion rules

5. JVM:
- Types: int, long, float, double
- No bool; minimal support for byte and short
- Mapped to actual hardware by the “JVM Program”

6. Go: Many non-mixable types, implemented as a library
- int8, int16, int32, int64
- uint8, uint16, uint32, uint64
- includes also type aliases

byte is uint8
rune is int32
int either
- int32, or

19. int is a synonym for signed int. short is a synonym for signed short, but char is not a synonym for signed char, nor for unsigned char
• int64
• uint again,
• uint32, or
• uint64, but,
• same size as int

7. Newspeak/Mock: Unbounded integers

• Integer/big maps to machine word
• in case of overflow, switches to long word
• in case of further overflow switches to an array of long words
• in case of even further overflow double the array size

---
183. Summary: “Integer” in various languages

Each language takes its own special perspective of the underlying hardware

**Pascal**  We know nothing of the hardware; we do not care about the hardware; so let’s assume it has a good enough integer

**BCPL**  Worship the unknown, mysterious and almighty Newspeak; programmer must cope with machine words.

**C/C++**  Squeeze the maximum out of hardware, whatever it is

**Java**  Let’s invent our own hardware!

**Go**  Yes, both C and C++ have failed; but we will do it better with predefined types.

**Newspeak**  Hardware is just an implementation detail; we want \( \mathbb{Z} \), and \( \mathbb{Z} \) we shall have!

---
3.5.4 More on language design

184. Force III–Language: complexity and stability

*Design is generally easy; Good design is difficult*

• PL specification complexity
• PL implementation complexity
• PL stability

**Example: Primitive Types of C**

1978 K&R language release;
1988 ANSI C first standard; new type “long double” that few people heard of at the time
1999 C 99:

• newly born, yet severely brain damaged “bool” type
• cannot add “bool” reserved identifier (defined in an include file as \_Bool)
• cannot add “complex” reserved identifier (_Complex instead)
• cannot add “I” reserved identifier (defined in an include file)
• new type long long int

---
185. “Simple” conversion rules of Java

19 widening operations 22 narrowing operations

**Figure 3.5.3: Java widening casts**

- Preserve magnitude
- May lose accuracy

**Figure 3.5.4: Java narrowing casts**

- magnitude and accuracy may be lost
- short and char can be “narrowed” to each other.

• 7 pseudo-numerical primitive types
• \( \binom{4!}{2} = 42 \) possible conversions
• JAVA defines 41 = 19 + 22 = 98% out of these.
• In C, \( 1 + 2 \times 5 + 2 \times 3 \approx 17 \) numerical primitive types; 272 possible conversions.

---
3.5.5 Real numbers

186. Who needs real numbers

Real Programmers don’t Use Reals?!

**Some-times**  No, or very few reals, e.g., in O/S programming

**Other-times**  You must have real numbers

Reals come in three varieties

1. **Infinite precision.** “real” real numbers, such as \( \pi \) and \( \sqrt{2} \), etc., used for symbolic computation
2. **Fixed point.** for representing currency, weights, distances, and the such
3. **Floating point.** for most scientific applications

---
187. Fixed point & infinite precision

**Fixed point:**

• Minimal support by modern hardware
• No support by most modern PLs
• Usually implemented, if necessary, in a library
Infinite precision:

- No support by modern hardware
- No support by most modern PLs
- Newspeak support infinite precision for integers and reals, including support for transcendental numbers.
- Can be implemented, if necessary, in a library

188. Standards for floating point numbers

- When you do need real numbers, you cannot live without them.
- Standards (of IEEE\textsuperscript{20}) for representing reals as floating points come to the rescue.

IEEE 754 interchange formats

<table>
<thead>
<tr>
<th>Width</th>
<th>Binary</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>binary16\textsuperscript{21}</td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>binary32\textsuperscript{22}</td>
<td>decimal32</td>
</tr>
<tr>
<td>64</td>
<td>binary64\textsuperscript{23}</td>
<td>decimal64</td>
</tr>
<tr>
<td>128</td>
<td>binary128\textsuperscript{24}</td>
<td>decimal128</td>
</tr>
</tbody>
</table>

Table 3.5.3: IEEE standards for representing numbers in floating point format

189. Other standards for representing floating point numbers

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Width</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 1130</td>
<td>32</td>
<td>n/a</td>
</tr>
<tr>
<td>IBM 1130</td>
<td>40</td>
<td>n/a</td>
</tr>
<tr>
<td>IBM/360</td>
<td>32</td>
<td>HFP32\textsuperscript{25}</td>
</tr>
<tr>
<td>IBM/360</td>
<td>64</td>
<td>HFP64</td>
</tr>
<tr>
<td>IBM/370</td>
<td>128</td>
<td>HFP128</td>
</tr>
<tr>
<td>X86 / X86-64</td>
<td>80\textsuperscript{26}</td>
<td>x86 EPF\textsuperscript{27}</td>
</tr>
</tbody>
</table>

Table 3.5.4: Some other (mostly obsolete) standards for floating points

190. Mantissa, exponent, sign

- **Sign**:
  + \[ +0.2245915772 \times 10^2 \] \hspace{1cm} (3.5.2)
  - First digit is never zero:
    \[ 0.1 < m \leq 1 \] \hspace{1cm} (3.5.6)
  - In binary base, first digit is always 1
    \[ 0.5 < m \leq 1 \] \hspace{1cm} (3.5.7)

- **Exponent**: 2
  \[ +0.2245915772 \times 10^2 \] \hspace{1cm} (3.5.3)

- **Mantissa**: 2245915772
  \[ 22 \text{ 45915772} \] \hspace{1cm} (3.5.4)

- **Normalized mantissa** \( m \): \[ +0.2245915772 \times 10^2 \] \hspace{1cm} (3.5.5)

- Special **NaN** is “not a number”, value, e.g.,
  \[ \sqrt{-1} \equiv \text{NaN} \] \hspace{1cm} (3.5.10)

191. A taste of floating point representation

The IEEE 754 / binary32 format for floating point representation:

```
  + 0 1 0 0 1 1 0 0 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
  + 01001100111000000000000000000000

  Sign \[ 0 \] positive
  Exponent (8 bits)
  Mantissa (23 bits)

Four bytes word (32 bits)
```

Questions answered by standard

- bits allocated for each component?
- representation of signed exponent? (Two’s complement or +N, e.g., +128)
- eliminate first bit of mantissa?
- ...

192. Issues in floating point representation

Standard are complex; essential details include:

- is it always normalized?
- allowing subnormal numbers (i.e., numbers where the mantissa is not normalized)?
- Signed zeroes:
  \[ -0 \neq +0 \] \hspace{1cm} (3.5.8)
- Multiple infinities:
  \[ -\infty \neq +\infty \] \hspace{1cm} (3.5.9)

- Special **NaN** is “not a number”, value, e.g.,
  \[ \sqrt{-1} \equiv \text{NaN} \] \hspace{1cm} (3.5.10)
- :

\textsuperscript{20} Institute of Electrical and Electronics Engineers
3.5.6 The “character” primitive type

The ever changing answer,…

$2^5 = 32$ punched tapes, telegraph

$10 \times 6 = 60$ early PASCAL

$2^6 = 64$ BCD

$2^7 = 128$ ASCII

$2^8 = 256$ EBCDIC, Extended ASCII, ISO 8859-1 Western Europe, IBM’s Code Page 437 (all encodings are distinct)

$2^{16} = 65,536$ Unicode 1.0; 1992; extension of ASCII; JAVA and other PLs of that time

$2^{19} \ll 1,114,112 < 2^{20}$ Unicode 2.0; 1996; of which about 110,000 characters are used; $\approx 100$ scripts; zillion of symbols.

Table 3.5.5: The ancient IBM 704 BCD character encoding

<table>
<thead>
<tr>
<th>Code point(s)</th>
<th>Character(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0–9</td>
<td>“0”,…, “9”</td>
</tr>
<tr>
<td>10</td>
<td>unassigned</td>
</tr>
<tr>
<td>11,12</td>
<td>“#”, “$”</td>
</tr>
<tr>
<td>13–15</td>
<td>unassigned</td>
</tr>
<tr>
<td>16</td>
<td>“&amp;”</td>
</tr>
<tr>
<td>17–25</td>
<td>“A”,…, “I”</td>
</tr>
<tr>
<td>26–28</td>
<td>“+0”, “.”, “#”</td>
</tr>
<tr>
<td>29–31</td>
<td>unassigned</td>
</tr>
<tr>
<td>32</td>
<td>“”</td>
</tr>
<tr>
<td>33–41</td>
<td>“J”,…, “K”</td>
</tr>
<tr>
<td>42–44</td>
<td>“-0”, “$”, “*”</td>
</tr>
<tr>
<td>45–47</td>
<td>unassigned</td>
</tr>
<tr>
<td>48–49</td>
<td>“U”, “J”</td>
</tr>
<tr>
<td>50–57</td>
<td>“S”, “T”, “Z”</td>
</tr>
<tr>
<td>58–59</td>
<td>“+”, “-”, “%”</td>
</tr>
<tr>
<td>60–63</td>
<td>unassigned</td>
</tr>
</tbody>
</table>

Notice

- “+”, “!” are missing; most people would today would think these are essential.
- “#”, “$”: most people today do not even know the names of these
- “-0”, “+0”: do not exist today as characters
- “A”,…, “Z”: do not receive consecutive code points
- No lower/upper case distinction.

Baudot Code by Émile Baudot, 1874; “Telegraph Alphabet No. 2 (ITA2)” 1930; $2^5 = 32$ characters; not even sufficient for letters and digits.

CDC Display Code $\approx 1960$; used in CDC computers on which PASCAL was developed; no 6 bits per character; $2^6 = 64$ or $2^6 - 1 = 63$ distinct characters; no upper/lower case distinction

BCD (IBM’s Binary Coded Decimal) $\approx 1960$; 6 bits per character; $2^6 = 64$ distinct characters.

ASCII (American Standard Code for Information Interchange) $\approx 1960$; $2^7 = 128$ characters: 95 printable + 33 controls. (used until today!)

EBCDIC (IBM’s Extended Binary Coded Decimal Interchange Code) $\approx 1965$; 8 bits per character; but not all $2^8 = 256$ code points were used.

Unicode 1.0 1992

Unicode 2.0 1996

Mostly, Yes! but,… there are still pockets of resistance Windows/IBM Variations; still in common use; all use 8-bits

Windows-1250 European Languages; similar to ISO-8859-2

Windows-1252 Similar to ISO-859-1, but not identical

Windows-1255 Hebrew; almost compatible superset of ISO 8859-8

Windows-1256 Arabic; is not compatible with ISO-8859-6

Mac OS Character Encodings

- Code page 10000
- Code page 10004
- Code page 10006
- Code page 10007
- Code page 10017
- Code page 10029
- Code page 10079
- KanjiTalk
- Mac Icelandic encoding
- Mac OS Roman
- MacArabic encoding
- MacGreek encoding
- Macintosh Central European encoding
- Macintosh Cyrillic encoding
- Macintosh Ukrainian encoding
197. Challenges in PL design

Huh?


- Why should I care?
- Are all these historical details, acronyms & standards really interesting?
- Some standards are much more important than others.

ASCII  Windows-1250  Unicode 1.0  Unicode 2.0–

Tough decisions

Portability a character literal may not be supported by all architectures

Elegance support Unicode 2.0 with its weird number of bytes

Efficiency most characters in common use can be represented using a single byte.

198. Impact of character encoding on PLs I

- No upper/lower case distinction in PASCAL (CDC)
- Some versions of PASCAL (CDC) have “string” type which is ten characters long (60-bit word= 10 × 6 characters)

Figure 3.5.6: A short “string” type on ancient CDC computers

| Ten “bytes” word: 10 × 6 = 60 bits / 10 characters |
| c₀  | c₁  | c₂  | c₃  | c₄  | c₅  | c₆  | c₇  | c₈  | c₉  |
| 6 bits | 6 bits | 6 bits | 6 bits | 6 bits | 6 bits | 6 bits | 6 bits | 6 bits |

- C treats primitive type char as a small integer
- About half of Unicode is not readily supported by JAVA
- Writing non-ASCII characters is not trivial in C/C++

199. Impact of character encoding on PLs II

- π and α are legal identifier names in JAVA
- C/C++ do not say whether char is
  - signed char, or
  - unsigned char
- Since some codes (e.g., European ISO-646) use certain character positions (e.g., “[” , “{”) for letters, C++ allows trigraph sequences, e.g.,
  - “??(” for “[”,
  - “??=” for “#”
- Programming in JAVA for Windows or MacOS can be challenging

3.5.7 Strings as atomic types

Strings are...

- A sequence of characters.
- Useful data type.
- Supported in one way or another by all modern PLs.

Issues:

- Atomic (as in ICON) or composite (as in PASCAL) type?
- Fixed length (as in COBOL) or variable length (as in C)?
- What string operations are supported?
- String literals? Delimiters or quotes?
- Are characters strings of length one?

200. Strings

ML  atomic type of any length. Operations: equality test, concatenation, decomposition are built-in.

Pascal  an array of characters.

Most trivial string operations require non-trivial programming.

Ada  as in PASCAL

C  as in PASCAL, but with string literals

Bash  and other scripting languages: full blown type.

Java  standard library implementation.

C++  standard library implementation.

3.5.8 Summary

201. Strings in various PLs

- Atomic type
- Primitive type
- Classification of types: Numeric, Non-Numeric, Semi-Numeric, Ordered, Unordered
- Concerns in selection of primitive types: portability, efficiency, intended use, complexity, stability
- Encoding: ASCII, Unicode, Windows-1250
- Encodings of floating point numbers
References

- 1’s complement
- 2’s complement
- Ada
- arbitrary precision
- ASCII
- Bash
- Baudot code
- BCD
- BCPL
- blintzes
- byte
- CDC Display Code
- character encoding
- Cobol
- codepage 437
- CPU architectures
- crêpe
- deprecation
- EBCDIC
- enumerated type
- Excel
- extended ASCII
- extended precision
- fixed point
- FORTRAN
- ICON
- IEEE floating point
- IEEE
- ISO-8859-1
- ISO-8859-2
- JAVA
- JVM
- pancake
- portability
- primitive data type
- Smörgåsbord
- SNOBOL
- string in C
- string literals
- taxonomy
- NEWSPEAK
- string in C++
- trigraphs in C++
- type string
- Unicode
- Windows-1250
- Windows-1252
- word

Exercises

1. Why should we expect languages that support Unicode letters in identifier names to distinguish between lower and upper case?
2. How does C++ support Unicode? Which version? How?
3. Modern language allow Unicode letters in identifier names. Why?
4. Why is the support of floating point numbers in MOCK brain-damaged?
5. Which is the missing conversion in JAVA? Why is it missing?
6. What are far pointers?
7. Determine whether type char in JAVA is numerical?
8. Writing non-ASCII characters is not trivial in C/C++
9. What are the literals of enumerated types?
10. What are the atomic types of AWK? PASCAL?
11. How come there are no conversions from- or to- atomic type boolean in JAVA?
12. Why does PASCAL fail to distinguish between lower and upper case letters?
13. Are all numerical types ordinal? If yes, explain why. If no, provide a counter example.
14. How come C++ does not say whether char is signed char, or unsigned char?
15. Is the type “string” in C++ a primitive type?
16. Are any digraphs used in JAVA? Explain the process by which you have reached your conclusion.
17. Determine whether type bool in JAVA ordered or not.
18. Does C include trigraphs? Which?
19. Are all ordinal types numerical? If yes, explain why. If no, provide a counter example.
20. Ancient versions of PASCAL had “string” type (called \texttt{alpha}) which was ten characters long. Why this particular magic number?

21. Languages that rely on character encoding other than ASCII and Unicode are rare. Why?

22. “About half of Unicode is not readily supported by Java”. Huh? How come? And why is this not such a big deal?

23. “Enumeration can be viewed as a combination of Unit, branding and disjoint union”. Explain, and demonstrate with ML. Why wouldn’t this work in C/C++?

24. Can you explain why ancient PASCAL used a sequence of two characters to delimit comments?

25. Why should programming in Java for Windows/MacOS be challenging?

26. Why does PASCAL fail to distinguish between lower and upper case letters?

27. Why is that PLs rarely offer literals for compound types?

28. Given is a type \( T \). If no other types can be found “within” \( T \). Does this mean that \( T \) is atomic?

29. BCPL included an equivalent of trigraphs. Why?

30. Why is the support of fixed point numbers in PASCAL gone? Why?

31. Give an example of a programming language offers literals for that compound type. If no, provide a counter example.

32. Are all ordered types numerical? If yes, explain why. If no, provide a counter example.

33. What is the “

34. Determine whether type \texttt{bool} in Java is ordered.

35. Given is a primitive type \( T \). Does this necessarily mean that other types can/cannot be found “within” \( T \)? How would your answer change if it is known that type \( T \) is atomic?

36. Why was it so easy to support character sets in ancient Pascal? How was set union and intersection implemented?

37. Write a program to test if Go uses type aliases or type branding in its management of machine types.

38. What’s the difference between a “trigraph” and a “di-graph”?

39. Why do C/C++ treat primitive type \texttt{char} as a small integer?

40. “Enumeration can be viewed as a combination of Unit, branding and disjoint union”... Explain why wouldn’t this work in C/C++?

41. Suppose that a certain hardware represents both an integer and a real number using precisely 42 bits. Explain why is it that precision must be lost in a conversion of an integer to real.

---

**List of definitions**

3.5.1 The values’ universe

- Every PL has a set of values, e.g., integers, tuples, records, functions,...
- Running a program of the PL, amounts to manipulation of members of this set.
- The values’ set is also called the universe of values.

3.5.1 Expression An expression is a part of a program whose evaluation during computation outcomes with a value.

3.5.1 Type error [on a single value] A program commits a type error on a value \( v \in T \) if it makes an attempt to manipulate \( v \) in a way which is inconsistent with \( T \).

3.5.1 Type error [on multiple values] A program commits a type error on values

\[\forall v_1, \ldots, v_n \in T, \ n \geq 1\]

if it makes an attempt to manipulate \( v_1, \ldots, v_n \) in a way which is inconsistent with \( T_1, \ldots, T_n \).

3.5.1 Pseudo type error A program commits a pseudo type error on a value \( v \in T \) if it makes an attempt to manipulate \( v \) in a way

- which is, in general, legal for \( T \).
- yet, is illegal for the particular \( v \in T \).

3.5.1 Type punning type punning is the power to interpret machine representation of \( v \) in a way which is inconsistent with \( v \)

3.5.1 Overloading An overloaded term is a term that has multiple meanings, which may, but also may not be related.

3.5.1 Identifier overloading An identifier or operator is said to be overloaded if it simultaneously denotes two or more distinct nameables

3.5.1 Power set type constructor If \( T \in \mathbb{T} \) is a type, then so is \( \wp T \), its power set comprising all subsets of \( T \), i.e.,

\[\wp T = \{ T' \mid T' \subseteq T \}\]

3.5.1 Cartesian product type constructor If \( T_1 \) and \( T_2 \) are types, their Cartesian product is a type denoted by \( T_1 \times T_2 \); values of \( T_1 \times T_2 \) are

\[T_1 \times T_2 = \{ \langle v_1, v_2 \rangle \mid v_1 \in T_1; v_2 \in T_2 \}\]

3.5.1 Integral exponentiation type constructor For a type \( T \in \mathbb{T} \) and \( n \in \mathbb{N} \), the integral exponentiation of \( T \) to the power of \( n \), \( T^n \), is defined by

\[T^n = T \times \cdots \times T \]
3.5.1 **The Unit type** The Unit type is $T^0$, where $T$ is some type; alternatively, Unit is a Cartesian product of zero types.

3.5.1 **Branding** If $T \in \mathbb{T}$ is a type and $\ell \in \mathbb{I}$ is label then $\ell(T)$ is the $\ell$ brand of $T$ where,

$$\ell(T) = \{ (\ell, v) | v \in T \}.$$

3.5.1 **Record type constructor** Let

$$\ell_1, \ldots, \ell_n \in \mathbb{I},$$

for $n \geq 0$ be unique labels. Let

$$T_1, \ldots, T_n$$

be types. Then,

$$\{\ell_1 : T_1, \ldots, \ell_n : T_n\}$$

is the record type induced by the labels $\ell_1, \ldots, \ell_n$ and types $T_1, \ldots, T_n$.

3.5.1 **Union type constructor**? If $T_1, T_2 \in \mathbb{T}$ are types, then so is their union, $T_1 \cup T_2$.

3.5.1 **Choice type** If $T_1, T_2 \in \mathbb{T}$ are types, then so is their disjoint union, $T_1 + T_2$, defined by the set

$$T_1 + T_2 = \ell_1(T_1) \cup \ell_2(T_2)$$

and where $\ell_1, \ell_2 \in \mathbb{I}$, $\ell_1 \neq \ell_2$ are some labels.

3.5.1 **Enumerated type constructor** If $\ell_1, \ell_2, \ldots, \ell_n \in \mathbb{I}$, $n \geq 1$ are labels, then

$$\{\ell_1, \ell_2, \ldots, \ell_n\}$$

is an enumerated type, whose values are

$$\ell_1, \ell_2, \ldots, \ell_n$$

3.5.1 **The Bottom type** Type None, also known as Bottom, and often denoted $\bot$, is the empty set, i.e.,

$$\text{None} \equiv \text{Bottom} \equiv \bot \equiv \emptyset.$$

3.5.1 **The Any type** Type Any, also known as All, or Top, and often denoted $\top$, is the universal set, i.e., for a language $\mathcal{L}$ with values’ universe $\mathcal{V}_\mathcal{L}$,

$$\text{Any} \equiv \top = \mathcal{V}_\mathcal{L}.$$

3.5.1 **Mapping** A mapping (also called function) from a set $S$ to a set $T$ is a set

$$m \subset S \times T,$$

that associates precisely one value of $T$ with each value of $S$, i.e.,

$$\forall s \in S : |\{ t | (s, t) \in m \}| = 1$$

3.5.1 **Partial mapping** We say that the set

$$m \subset S \times T$$

is a partial mapping (partial function), from set $S$ to set $T$ if it never associates more than value of $T$ with any value of $S$, i.e.,

$$\forall s \in S : |\{ t | (s, t) \in m \}| \leq 1$$

3.5.1 **Mapping (partial mapping) type constructor** Let $T$ and $S$ be types. Then, (partial) mapping from $S$ to $T$, denoted $S \rightarrow T$ (sometimes also $T^5$) is

$$S \rightarrow T = \{ m | m \text{ is a (partial) mapping from } S \text{ to } T \}.$$

3.5.1 **Recursive type definition**

- Let $T_1, \ldots, T_m$ be some fixed types.
- Let $\tau_1, \ldots, \tau_n$ be type unknowns.
- Let $E_1, \ldots, E_n$ be type expressions involving types $T_1, \ldots, T_m$ and unknowns $\tau_1, \ldots, \tau_n$.
- Then, the system of equations

$$\tau_1 = E_1(T_1, \ldots, T_m, \tau_1, \ldots, \tau_n)$$

$$\vdots$$

$$\tau_n = E_n(T_1, \ldots, T_m, \tau_1, \ldots, \tau_n)$$

defines new types $\tau_1, \ldots, \tau_n$ as the “minimal” solution of this system.

3.5.1 **Tag** Tag is the mechanism for storing the selection made in a choice type along with the value associated with the choice.

3.5.1 **Operations on pointers Construction**

- Create a null pointer.
- Create a pointer to a “stored” value.

**Tag testing** Determine whether a pointer is null or not.

**Projection** If the pointer is not null, extract value.

3.5.1 **Type error** A program commits a type error if it makes an attempt to:

- manipulate $v$ in a way which is inconsistent with $T$.
- interpret machine representation of $v$ in a way which is inconsistent with $T$.

for some type $T$ and a value $v \in T$.

3.5.1 **Primitive type** A type which is both atomic and builtin, i.e., is not programmer defined, is called a primitive type.

---

**List of tables**

**Unit 3.1 Value systems**

**Subunit 3.1.2 Semantics of $S$-expressions?**

- 3.1.1 Examples of using the “$c[ad]+r$” syntactic sugar

- 3.1.2 Syntactical differences between functions & operators

---

40
Unit 3.2 Introduction to types
Subunit 3.2.1 Types as sets
3.2.1 A set of values which does not make a type and a set that does .................. 8
Subunit 3.2.3 Type determines semantics: the overloading case study
3.2.2 Overloaded meanings of terms in the Groucho Marx riddle ......................... 10
Unit 3.3 The type constructors of Mock
Subunit 3.3.11 Summary
3.3.1 Taxonomy of the type constructors of MOCK 20
Unit 3.5 Atomic types
Subunit 3.5.1 Taxonomy of atomic types
3.3.1 Classification of primitive types ............. 32
3.5.2 More classification: unordered, ordered & ordinal types .......................... 32
Subunit 3.5.5 Real numbers
3.5.3 IEEE standards for representing numbers in floating point format ............... 35
3.5.4 Some other (mostly obsolete) standards for floating points ....................... 36
Subunit 3.5.6 The “character” primitive type
3.5.5 The ancient IBM 704 BCD character encoding 37

List of figures
Unit 3.1 Value systems
Subunit 3.1.1 Symbolic values
3.1.1 The CONS record .......................... 3
3.1.2 The list (a b c d) in binary tree representation ........................................ 3
Subunit 3.1.2 Semantics of S-expressions?
3.1.3 Understanding “c[ad]+r” syntactic sugar ................................................. 4
3.1.4 The many ways for presenting the symbolic values of MATHEMATICA ........... 5
Unit 3.2 Introduction to types
Subunit 3.2.1 Types as sets
3.2.1 Visualizing the type system ............................................................... 7
Subunit 3.2.3 Type determines semantics: the overloading case study
3.2.2 Unraveling the Marx riddle .................................................. 10
3.2.3 Archie Bunker explains to Edith Bunker how context is used to resolve the ambiguity of the three overloaded meanings of the word “Shalom” in Hebrew ........................................ 11
Unit 3.3 The type constructors of Mock
Subunit 3.3.10 Recursive type constructor
3.3.1 Using MATHEMATICA to solve the quadratic equation \( \tau = 1 + \mathbb{Z}^2 \) for \( \tau \), the recursive binary tree type .................................................. 19
3.3.2 Distinct topologies of binary trees with with \( n \) nodes; \( n = 0, 1, 2, 3 \); each node stores an integer \( i \in \mathbb{Z} \) .................................................. 19
Subunit 3.3.11 Summary
3.3.3 Taxonomy of the type constructors of MOCK 19
Unit 3.4 Type constructors in actual PLs
Subunit 3.4.5 Tags in concrete PLs
41

List of equations
(3.1.1) The type system is a set of sets of values ........ 1

\( \mathbb{P} \subseteq \mathbb{Q} \)

(3.1.2) Policy for representation of values on hardware 1

\( \mathbb{P} : \mathbb{Q} \mapsto \{ M_1, M_2, \ldots \} \)

3.2.1 Types as sets
(3.2.1) A type of PL “is” a set of values .............. 6

\( \forall \tau \in \mathbb{P} : \tau \subseteq \mathbb{Q} \).

(3.2.2) The set of types of a PL .......................... 6

\( \mathbb{P} \subseteq \mathbb{Q} \).

(3.2.3) The set of types of a value .......................... 6

\( \text{types}(v) = \{ T \in \mathbb{Q} | v \in T \} \)

(3.2.4) Some values have more than one type: ...... 6

\( \forall v : \text{types}(v) > 0 \)

(3.2.5) Some values have more than one type: ...... 6

\( \exists v : \text{types}(v) > 1 \)

(3.2.6) Typically, values with more than one type, have infinitely many types: .................. 6

\( \text{types}(v) > 1 \Leftrightarrow \text{types}(v) = \infty \)

(3.2.7) Infinitely many types of 0 in C .................. 7

\( T \in \mathbb{T} \Rightarrow \text{“0” } \in T^* \)

3.3.1 Power sets
(3.3.1) Power set type constructor .......................... 12

\( \mathbb{P} \mathbb{T} = \{ T' | T' \subseteq T \} \).

(3.3.2) Alternative notation for power sets .......................... 12

\( \mathbb{P} \mathbb{T} = 2^T \)
(3.3.3) Cardinality of type constructed with the power set type constructor ........................................... 12
\(|\mathcal{P}(\mathcal{T})| = 2^{\mathcal{T}}|

(3.3.4) The empty set is a value of all power sets .................. 13
\(\forall \mathcal{T} : \emptyset \in \mathcal{P}(\mathcal{T})\)

(3.3.5) A unary value constructor for power sets .................. 13
\(\forall v \in \mathcal{T} : \{v\} \in \mathcal{P}(\mathcal{T})\)

(3.3.6) An n-ary value constructor for power sets .................. 13
\(\forall v_1, v_2, \ldots, v_n \in \mathcal{T} : n \geq 1 \Rightarrow \{v_1, v_2, \ldots, v_n\} \in \mathcal{P}(\mathcal{T})\)

3.3.2 Cartesian product

(3.3.7) Cartesian product type constructor .................. 13
\(\mathcal{T}_1 \times \mathcal{T}_2 = \{(v_1, v_2) | v_1 \in \mathcal{T}_1; v_2 \in \mathcal{T}_2\}\)

(3.3.8) Cardinality of type created by Cartesian product type constructor ........................................... 13
\(|\mathcal{T}_1 \times \mathcal{T}_2| = (|\mathcal{T}_1| \times |\mathcal{T}_2|)\)

(3.3.9) Operator for composing a value of a product type ........................................... 13
\(\forall v_1 \in \mathcal{T}_1, v_2 \in \mathcal{T}_2 \Rightarrow (v_1, v_2) \in \mathcal{T}_1 \times \mathcal{T}_2\)

(3.3.10) Operators for decomposing a value of a product type ........................................... 13
\(\forall v \in \mathcal{T}_1 \times \mathcal{T}_2 \Rightarrow v\#1 \in \mathcal{T}_1 \land v\#2 \in \mathcal{T}_2\)

3.3.3 Integral exponentiation

(3.3.11) Integral exponentiation type constructor ................. 14
\(\mathcal{T}^n = \mathcal{T} \times \cdots \times \mathcal{T}\) (n times)

(3.3.12) Cardinality of type created by integral exponentiation type constructor ........................................... 14
\(|\mathcal{T}^n| = (|\mathcal{T}|)^n\).

3.3.4 Unit type

(3.3.13) Cardinality of type Unit ........................................... 14
\(|\text{Unit}| = 1\)

(3.3.14) Type Unit as a singleton set ........................................... 14
Unit = \{\}.

(3.3.15) Bits required to represent a value of type Unit ................. 14
\(|\log_2(\text{Unit})| = \log_2(1) = 0\).

3.3.5 Branding

(3.3.16) Branding type constructor ........................................... 14
\(\ell(\mathcal{T}) = \{\langle \ell(v), v \rangle | v \in \mathcal{T}\}\).

(3.3.17) The first rule of branding ........................................... 14
\(\forall \ell \in \ell : \mathcal{T} \neq \ell(\mathcal{T})\)

(3.3.18) The second rule of branding ........................................... 14
\(\forall \ell_1, \ell_2 \in \ell : \ell_1 \neq \ell_2 \Rightarrow \ell_1(\mathcal{T}) \neq \ell_2(\mathcal{T})\)

(3.3.19) Operator to create a value of \(\ell(\mathcal{T})\) from \(v \in \mathcal{T}\). ........................................... 14
\(\forall v \in \mathcal{T} \Rightarrow \ell(v) \in \ell(\mathcal{T})\)

(3.3.20) Operator to extract the \(T\) value from a value of \(\ell(\mathcal{T})\). ........................................... 14
\(\ell(v) \in \mathcal{T} \Rightarrow \ell(v)\#\ell \in \mathcal{T}\)

3.3.6 Records

(3.3.21) Record types as product of branded types ................. 15
\(\forall \ell_1 : \mathcal{T}_1, \ldots, \ell_n : \mathcal{T}_n : (\ell_1(\mathcal{T}_1) \times \cdots \times \ell_n(\mathcal{T}_n))\)

(3.3.22) Composition operator to create a value of record type ........................................... 15
\(\forall v_1 \in \mathcal{T}_1, \ldots, v_n \in \mathcal{T}_n : \{\ell_1 : v_1, \ldots, \ell_n : v_n\} \in \mathcal{T}_1 \times \cdots \times \mathcal{T}_n\)

(3.3.23) Decomposition operator to elicit a field from a record type ........................................... 15
\(\forall i, 1 \leq i \leq n : \{\ell_1 : v_1, \ldots, \ell_n : v_n\}\#\ell_i = v_i\)

3.3.7 Disjoint union

(3.3.24) Disjoint union type constructor ........................................... 15
\(\mathcal{T}_1 + \mathcal{T}_2 = \ell_1(\mathcal{T}_1) \cup \ell_2(\mathcal{T}_2)\)

(3.3.25) Cardinality of type created by disjoint union ........................................... 15
\(|\mathcal{T}_1 + \mathcal{T}_2| = |\mathcal{T}_1| + |\mathcal{T}_2|\)

(3.3.26) Enumerated type as a disjoint union of branded Units ........................................... 15
\(\forall \ell_1, \ell_2, \ldots, \ell_n : \ell_1(\text{Unit}) + \ell_2(\text{Unit}) + \cdots + \ell_n(\text{Unit})\)

(3.3.27) Making values of a choice type ........................................... 15
\(v_1 \in \mathcal{T}_1 \Rightarrow \ell_1(v_1) \in \mathcal{T}_1 + \mathcal{T}_2\)

(3.3.28) Testing choice types ........................................... 15
\(v \in \mathcal{T}_1 + \mathcal{T}_2 \Rightarrow v\#1 = \begin{cases}\text{true} & v = \ell_1(v_1), v_1 \in \mathcal{T}_1 \\ \text{false} & v = \ell_2(v_2), v_2 \in \mathcal{T}_2. \end{cases}\)

3.3.8 Type None and type Any

(3.3.29) The empty type ........................................... 16
\(\text{None} \equiv \text{Bottom} \equiv \bot \equiv \emptyset\).

(3.3.30) Size of the empty type ........................................... 16
\(|\bot| = 0\)

(3.3.31) The universal type ........................................... 16
\(\text{Any} \equiv \top = \bigvee \mathcal{X}\).

(3.3.32) Cartesian product of arbitrary type and \(\bot\) ........................................... 16
\(\forall \mathcal{T} \in \mathcal{T} : \mathcal{T} \times \bot \subseteq \bot \)

(3.3.33) Disjoint union of arbitrary type with \(\bot\) and \(\top\) ........................................... 16
\(\forall \mathcal{T} \in \mathcal{T} : \downarrow + \mathcal{T} = \bot \)

3.3.9 Mapping types

(3.3.34) Full mapping ........................................... 16
\(\forall s \in S : |\ell(s, t) \in m| = 1\)

(3.3.35) Partial mapping ........................................... 16
\(\forall s \in S : |\ell(s, t) \in m| \leq 1\)

(3.3.36) Mapping type constructor ........................................... 16
\(S \rightarrow \mathcal{T} = \{m | m \text{ is a (partial) mapping from } S \text{ to } \mathcal{T}\}\).

(3.3.37) Power set type constructor as a kind of mapping ........................................... 16
\(\mathcal{P}(\mathcal{T}) \rightarrow \text{Boolean}\)

(3.3.38) Cardinality of type created by mapping type constructor ........................................... 16
\(|S \rightarrow \mathcal{T}| = |\mathcal{T}|^{|S|}\).
3.3.39 Mapping as exponentiation .......................... 16
\[ S \to T = T^S \]

3.3.40 Currying .................................................. 16
\[(S_1 \times S_2) \to T = S_1 \to (S_2 \to T)\]

3.3.41 Currying in algebra: power to product is power of power ........................................ 16
\[ T^{S_1 \times S_2} = (T^{S_1})^{S_2} \]

3.3.42 Integer range type constructor ............... 16
\[ n = 1, \ldots, n. \]

3.3.43 Integral exponentiation type constructor .... 16
\[ n \to \mathbb{R} = \mathbb{R}^n \]

3.3.44 Type Unit as special case of integer range . 16
\[ \text{Unit} = 1. \]

3.3.45 Type None as special case of integer range 16
\[ \text{None} = 0. \]

3.3.46 Euler’s identity, tying together the five fundamental mathematical constants ................. 16
\[ e^{i\pi} + 1 = 0. \]

3.3.47 Identity tying together \( \top, \perp, \) and Unit .... 16
\[ \top^\perp = 1. \]

3.3.10 Recursive type constructor ....................... 17

3.3.48 Recursive definitions type constructor ...... 17
\[ \tau_1 = E_1(T_1, \ldots, T_m, \tau_1, \ldots, \tau_n) \]
\[ \vdots \]
\[ \tau_n = E_n(T_1, \ldots, T_m, \tau_1, \ldots, \tau_n) \]

3.3.49 A recursive equation with only one type variable ........................................ 17
\[ \sigma = E(T_1, \ldots, T_m, \sigma) \]

3.3.50 Bottom up solution of recursive equation with only one type variable ..................... 17
\[ \sigma_0 = 0 = \perp \]
\[ \sigma_1 = E(T_1, \ldots, T_m, \sigma_0) = E(T_1, \ldots, T_m, \perp) \]
\[ \sigma_2 = E(T_1, \ldots, T_m, \sigma_1) = E(T_1, \ldots, T_m, E(T_1, \ldots, T_m, \perp)) \]
\[ \vdots \]
\[ \sigma_{n+1} = E(T_1, \ldots, T_m, \sigma_n) = E(T_1, \ldots, T_m, E(T_1, \ldots, T_m, \sigma_{n-1})) \]
\[ \vdots \]
\[ \sigma = \bigcup_{i=0}^{\infty} \sigma_i. \]

3.3.51 Recursive type equation for \( t \), the type of a pointer to a node in a doubly-linked-list .... 17
\[ t = (1 + T) = 1 + Z \times t \times t \]

3.3.52 Taylor expansion of the “negative” solution of the quadratic equation \( \tau = 1 + Z \tau^2 \) defining \( \tau \) ....... 18
\[ \tau = 1 + Z + 2Z^2 + 5Z^3 + 14Z^4 + 42Z^5 + \cdots \]

3.3.8 Mapping as functions and arrays