Lecture 12: Data Types (and Some Leftover ML)

COMP 524 Programming Language Concepts
Stephen Olivier
March 3, 2009

Based on slides by A. Block, notes by N. Fisher, F. Hernandez-Campos, and D. Stotts

Goals

- Introduce concepts pertaining to data types
- Examine the ML type system, polymorphism, and higher order functions
  - map, foldl, and foldr built-ins especially

Data Types

- Computers manipulate **sequences of bits**
- We manipulate **higher level data** (numbers, strings, etc.)
- **Data types** transform bits into higher level data

Data Types:

- Types provide **implicit context**
  - **Compilers can infer information**, so programmers write less code.
  - e.g., The expression **a+b** in Java may be adding two **integer**, two **floats** or two strings depending on **context**
- Types provide a set of **semantically valid operations**
  - Compilers can **detect semantic mistakes**
  - e.g., Python's list type supports append() and pop(), but complex numbers do not

Type Systems

- A type system consists of
  1. A mechanism to **define types** and **associate them with language constructs**
  2. A set of rules for “**type equivalence**,” “**type compatibility**,” and “**type inference**.”

Type Systems

- A type system consists of
    1. A mechanism to **define types** and **associate them with languag**
    2. A set of rules for "**type equivalence**," "**type compatibility**," and "**type inference**."

Discuss these in detail

Type Systems: Type Checking

- **Type Checking** is the process of ensuring that a program **obeys the language's type compatibility rules**
  - Strongly typed.
  - Weakly typed.

Strongly Typed

- **Strongly typed** languages **always detect type errors**
  - All **expressions and objects** must have a type
  - All operations must be applied in **appropriate type contexts**
- **Statically typed** languages are **strongly typed** languages in which **all type checking occurs at compile time**

Strongly Typed

Even FUNCTIONS!

- **Strongly typed** languages always **detect type errors**
  - All **expressions and objects** must have a type
  - All operations must be applied in **appropriate type contexts**
- **Statically typed** languages are **strongly typed** languages in which **all type checking occurs at compilee time**

Weakly Typed

- In **weakly typed** languages "anything can go"
  - Characteristic of assembly language
  - See also: Perl and earlier scripting languages
- On the other end of the spectrum, strongly typed languages don't allow implicit conversion

What is a type?

- Three points of view
  - **Denotational**: Set of values
  - **Constructive**: A type is “**built-in**” or “**composite**”
  - **Abstraction-based**: A type is an interface that defines a set of consistent operations

Denotation

- Under denotation, a **value has a given type if it belongs to a set**.
- An object has a type, **if its value is guaranteed to be in a certain set**.
- A set of values is called a **domain** (i.e., its type).
- Similar to **enum** in C

Built-in Types

- Built-in/primitive/elementary types
  - Mimic hardware units
  - e.g., boolean, character, integer, real (float)
- Implementation **varies** across languages
- Characters are **traditionally** one-byte quantitates using the ASCII character set

Built-in Types: Unicode

- Newer languages have built-in characters that support Unicode character sets
- **Unicode is implemented using two-byte quantities**.

Built-in Types: Unicode

- Newer languages have built-in characters that support Unicode character sets
- **Unicode is implemented using two-byte quantities**.

This is very important for moving legacy code.

Built-in Types: Numeric Types

- Most languages support **integers and floats**
  - (Their value range is implementation dependent)
- Some languages support other numeric types
  - Complex Numbers (e.g., Fortran, Python)
  - Rational Number (e.g., scheme, common Lisp)
  - Signed and Unsigned integers (e.g., C, Modula-2)
  - Fixed point Numbers (e.g., Ada, Cobol)
- Some languages distinguish numeric types depending on their precision.

Composite

- A composite type is created by **applying type constructors to simpler types**
  - Records
  - Structs
  - Arrays
  - Sets
  - Classes

Classification of Types: Enumerations

- **Enumerations** improve program readability and error checking.
- First introduced in Pascal (but also exist in C):
  - type weekday = (sun, mon, tue, wed, thu, fri, sat);
  - They are **defined in order**, so they can be used in enumeration controlled loops

Classification of Types: Subranges

- **Subranges** define a **valid range of values** for a variable.
    - e.g., Type test_score = 0..100;
- The improve **readability** and **error checking**

Classification of Types: Orthogonality

- Recall, **orthogonality** means that **all features behaves consistently**.
  - e.g., a=b always denotes **assignment**.
- This makes life much easier when reasoning about different types.

Type Checking

Now that we've discussed the basics of types, lets go back to **equivalence**, **compatibility** and **inference**.

Type Checking

- **Type Equivalence**: When are the **types of two values are the same**?
- **Type Compatibility**: Can a value of **A be used when type B is expected**?
- **Type Inference**: What is the **type of an expression**, given the type of the **operands**?

Type Checking

- **Type Equivalence**: When are the **types of two values are the same**?
- **Type Compatibility**: Can a value of **A be used when type B is expected**?
- **Type Inference**: What is the **type of an expression,** given the type of the **operands**?

Type Equivalence

- Type Equivalence is defined in terms of **structural** and **name equivalence**.

Structural Equivalence

- Two types are structurally equivalent if they have the same **components** put together **in the same way**

```
typedef struct{int a,b;} foo1
```

Equivalent!

```
typedef struct{
  int a,b;
}foo2
```

Structural Equivalence

- Two types are structurally equivalent if they have the same **components** put together **in the same way**

```
typedef struct{int a,b;} foo1
```

Equivalent?

Yes, in most languages.

```
typedef struct{
  int b;
  int a;
}foo2
```

Structural Equivalence

Equivalent...

```
typedef struct{
  char *name;
  char *addre;
  int age;
} student;
```

```
typedef struct{
  char *name;
  char *addre;
  int age;
} school;
```

... but probably not intentional.

Name Equivalence.

- **Name equivalence** assumes that **two definitions with different names are not the same**.
- Solves the “student-school” problem

Name Equivalence: Aliases

- Under name equivalence it is possible to define a new type via

```
TYPE new_type = old_type;
```

- Such a construction is called an **alias**.

```
TYPE new_type = old_type;
```

- Two ways to interpret an alias:
  - **Strict name equivalence**
    - **New_type** is a different type than **old_type**.
  - **Loose name equivalence**
    - **New_type** is the same type as **old_type**.

Problem with Loose

```
TYPE celsius_temp = REAL;
     farhen_temp = REAL;
VAR  c: celsius_temp;
     f: farhen_temp;
...
f:=c;(* probably should be an error*)
```

Type Conversion

- A value of one type **can be used in a context of another** type using **type conversion** or **type cast**

Converting Type Cast

- Under a **converting type cast**, the **underlying bits are changed**

C

```
int i;
float f= 3.4;
i = (int) f;
/* runtime */
```

Non-Converting Type Cast

- Under a **Non-converting type cast**, the **underlying bits are not altered.**

C

```
int i;
float f= 3.4;
i = *((int*) & f);
/* Compile time*/
```

Type Checking

- **Type Equivalence**: When are the **types of two values are the same**?
- **Type Compatibility**: Can a value of **A be used when type B is expected**?
- **Type Inference**: What is the **type of an expression,** given the type of the **operands**?

Type Compatibility

- Most languages **do not require type equivalence in every context**
- Two types **T and S are compatible** in Ada if any of the following conditions are true:
  - T and S are equivalent
  - T is a subtype of S
  - S is a subtype of T
  - T and S are arrays with the same number elements and same type of elements

Type Compatibility

- Type **coercion** allows a value of **one type to be used in a context that expects another**.

C

```
short int s;
unsigned long int l;
...
s=l;
```

Type Co

This makes the system type weaker.

- Type **co** … **type to be used in a context that expects another**.

C

```c
short int s;
unsigned long int l;
...
s=l;
```

Generic Reference Types

- It is often useful to have a **generic reference type** that can hold any type of object
  - in Java this is **Object**
  - In C and C++ this is **void ***

C

```
void* v;
int* i;
...
v=i;
```

Type Checking

- **Type Equivalence**: When are the **types of two values are the same**?
- **Type Compatibility**: Can a value of **A be used when type B is expected**?
- **Type Inference**: What is the **type of an expression**, given the type of the **operands**?

Type Inference

- Usually the type of the **overall expression is easy**.
- However, for **subranges** and **composite** objects is not so simple.

Subranges

```
type Atype = 0..20;
     Btype = 10..20;
var a: Atype;
    b: Btype;
...
a+b;
```

Pascal

What is the type of a+b?

Types in ML: Type Inference Extreme

- Full-blown type inference
- The “feel” of untyped declarations without losing the checks provided by strong typing
- Accommodates polymorphism:

```
fun fib n =
    let fun fib_helper f1 f2 i =
        if i = n then f2 else fib_helper f2 (f1 + f2) (i + 1)
    in
        fib_helper 0 1 0
    end;
```

- ML figures out that fib is a function that takes an integer and retains an integer through a series of deductions, usually starting with any literals

ML Type Correctness = Type Consistency

- The key to ML's type inference is the absence of inconsistency or ambiguity.
  - Functions whose type cannot be inferred by the operators or literals used will require explicit type declarations:

```
fun isquare x = x * x; (* Defaults to int -> int *)

fun rsquare x:real = x * x; (* real -> real *)
```

- But polymorphism is used where possible...

Polymorphism in ML

- Functions that do not use literals or type-specific operations in their definitions are recognized by the interpreter as polymorphic:

```
- fun twice f x = f (f x);
val twice = fn : ('a -> 'a) -> 'a -> 'a

- twice (fn x => x / 2.0) 1.0;
val it = 0.25 : real

- twice (fn x => x ^ "ee") "whoop";
val it = "whoopeeee" : string
```

Type Unification

- Part of ML's type inference is **unification** — composing or combining multiple types in a consistent manner
  - Example: `E1` has type `'a * int` and `E2` has type `string * 'b`
  - `if true then E1 else E2` is inferred as having type `string * int`
- Application for polymorphic operations on data structures
  - List manipulation orthogonal to type of list
  - Operations on user-defined data types
    - e.g. binary tree insertion, deletion, search
  - Higher order functions

Built-in Higher Order Functions: map

- map applies a given function to every element in the list

Is actually a curried function of type `('a -> 'b) -> 'a list -> 'b list`

- Format: `map function list`

```
- fun times2 x = x * 2.0;
val times2 = fn : real -> real
- map times2 [2.5,5.0,7.5];
val it = [5.0,10.0,15.0] : real list
```

- Can also use anonymous function:

```
- map (fn x => 2 * x) [1,2,3];
val it = [2,4,6] : int list
```

Built-in Higher Order Functions: foldr and foldl

- `foldr` combines elements of of a list using a given operation
  - Known in functional programming circles as reduce
- Again, a curried function
  - Type is `('a * 'b -> 'b) -> 'b -> 'a list -> 'b`
- Format: `foldr binary_function start_value list`

```
- foldr (op +) 0 [1,2,3,4];
val it = 10 : int
```

- op keyword before an operator gives the underlying function
  - e.g., can pass `(op <)` as an argument of type `int * int -> bool`

Built-in Higher Order Functions: foldr and foldl

- More examples:

```
- foldr op* 1.0 [2.0, 4.0];
val it = 8.0 : real

- foldr (op ^) "" ["abc","def","ghi"];
val it = "abcdefghi" : string
```

- `foldl` is a left-to-right version of `foldr`
  - Different results for operations like subtraction:

```
- foldl (op -) 0 [1,2,3,4]; (* 4-(3-(2-(1-0))) = 2 *)
val it = 2 : int

- foldr (op -) 0 [1,2,3,4]; (* 1-(2-(3-(4-0))) = ~2 *)
val it = ~2 : int
```

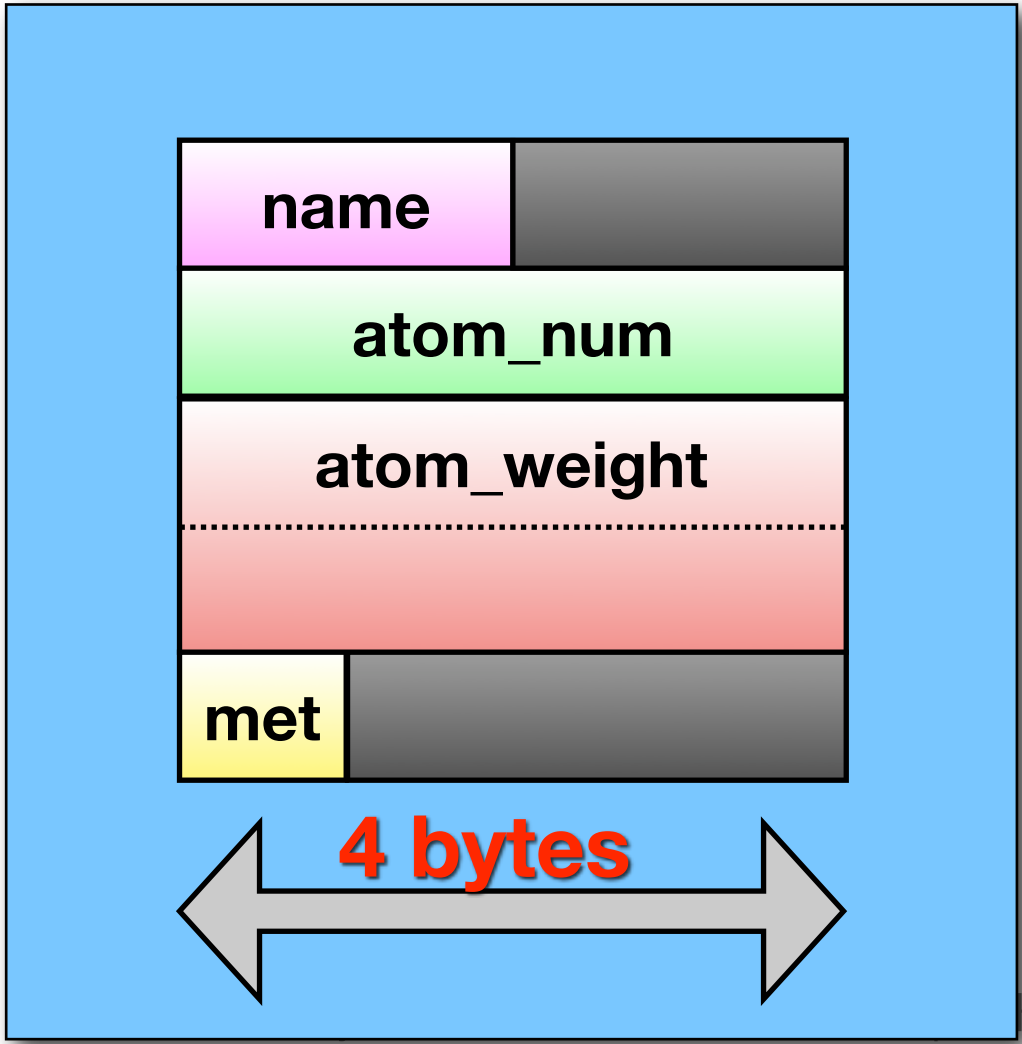
Records

- **Records** (structs in C and C++) allow **for a collection of related data to be manipulated together**.

```
struct foo{
  int a;
  int b;
}
```

Record: Memory Layout

- There may be **holes** in the allocation of memory

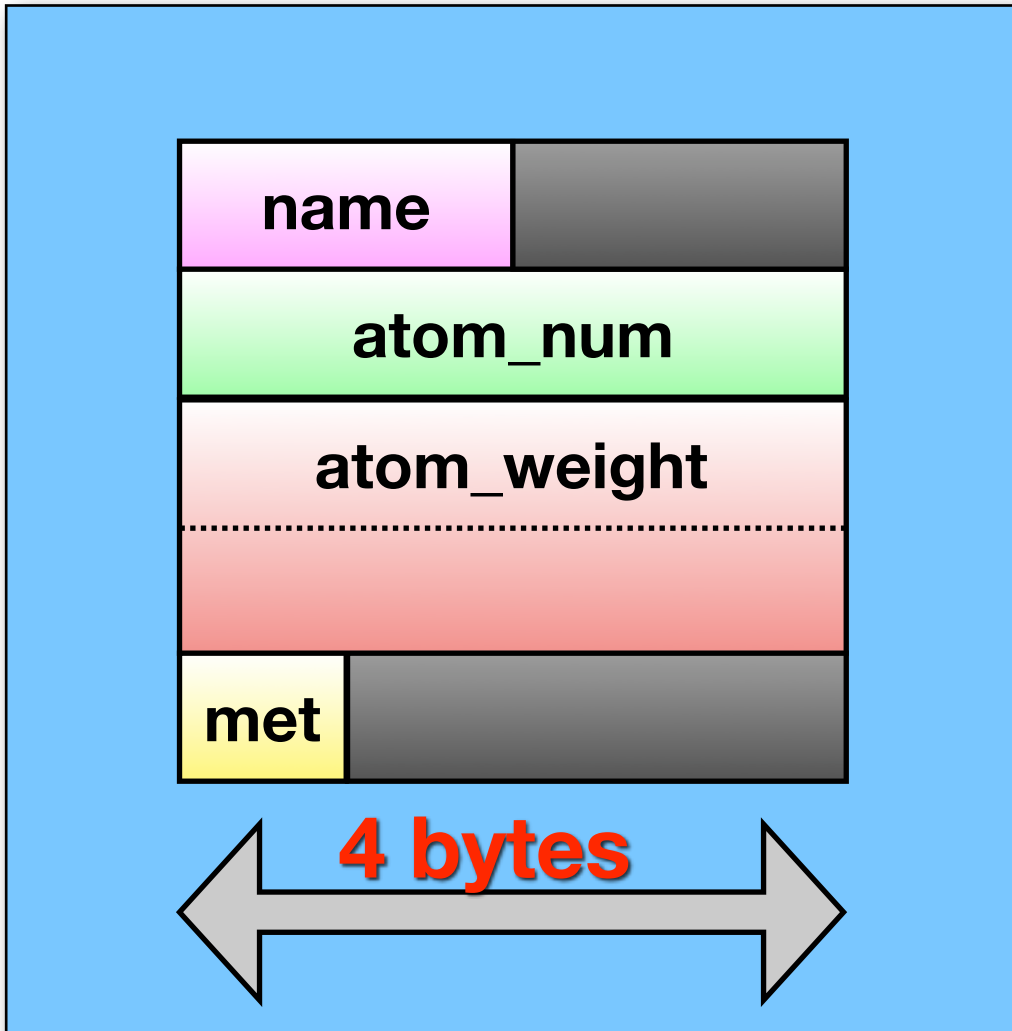



```
type ore = record
    name : two_char;
    atom_num: integer;
    atom_weight: real;
    met: Boolean;
end;
```

Record: Memory Layo

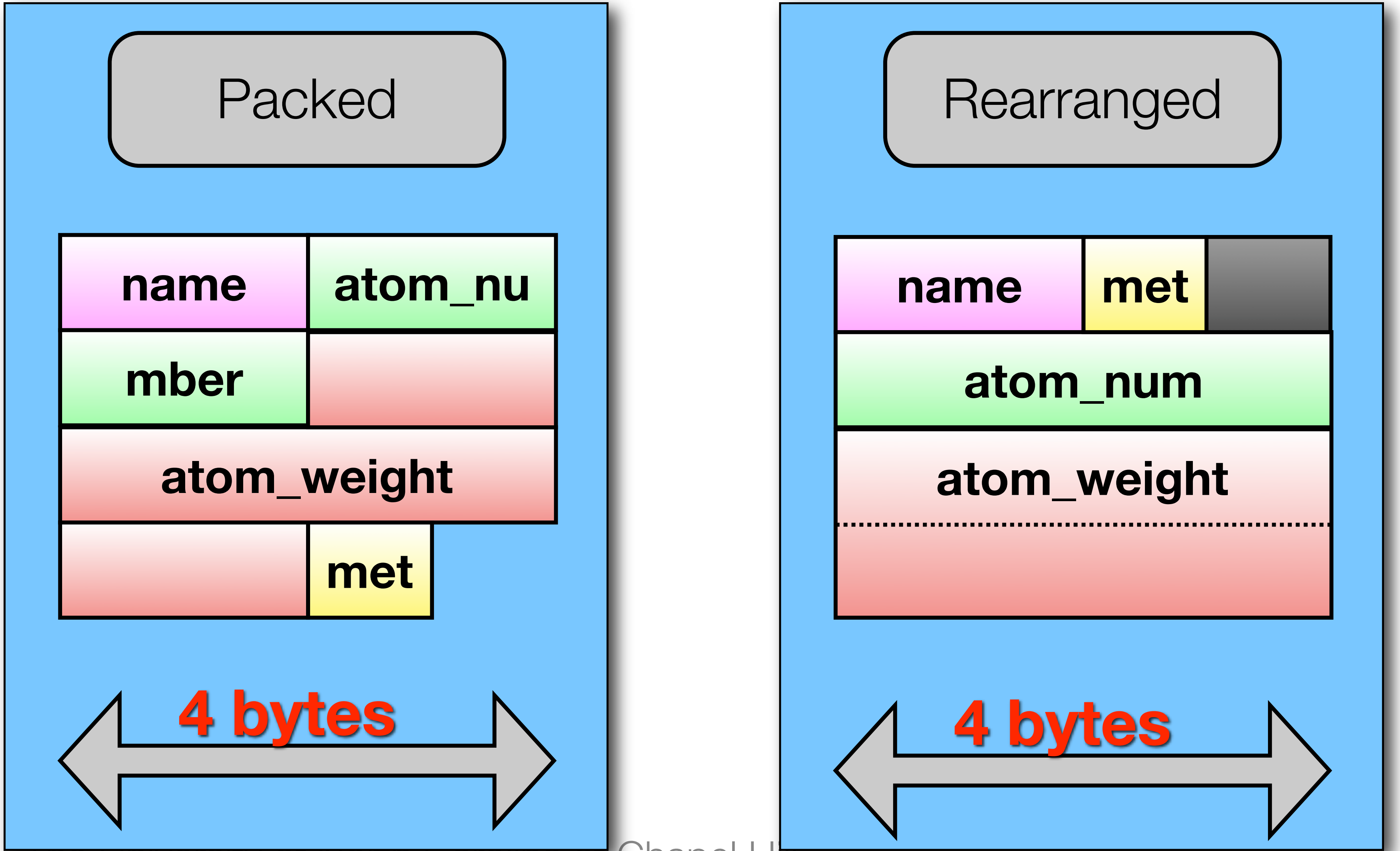
Holes waste space and complicate comparisons.

- There may be **holes** in

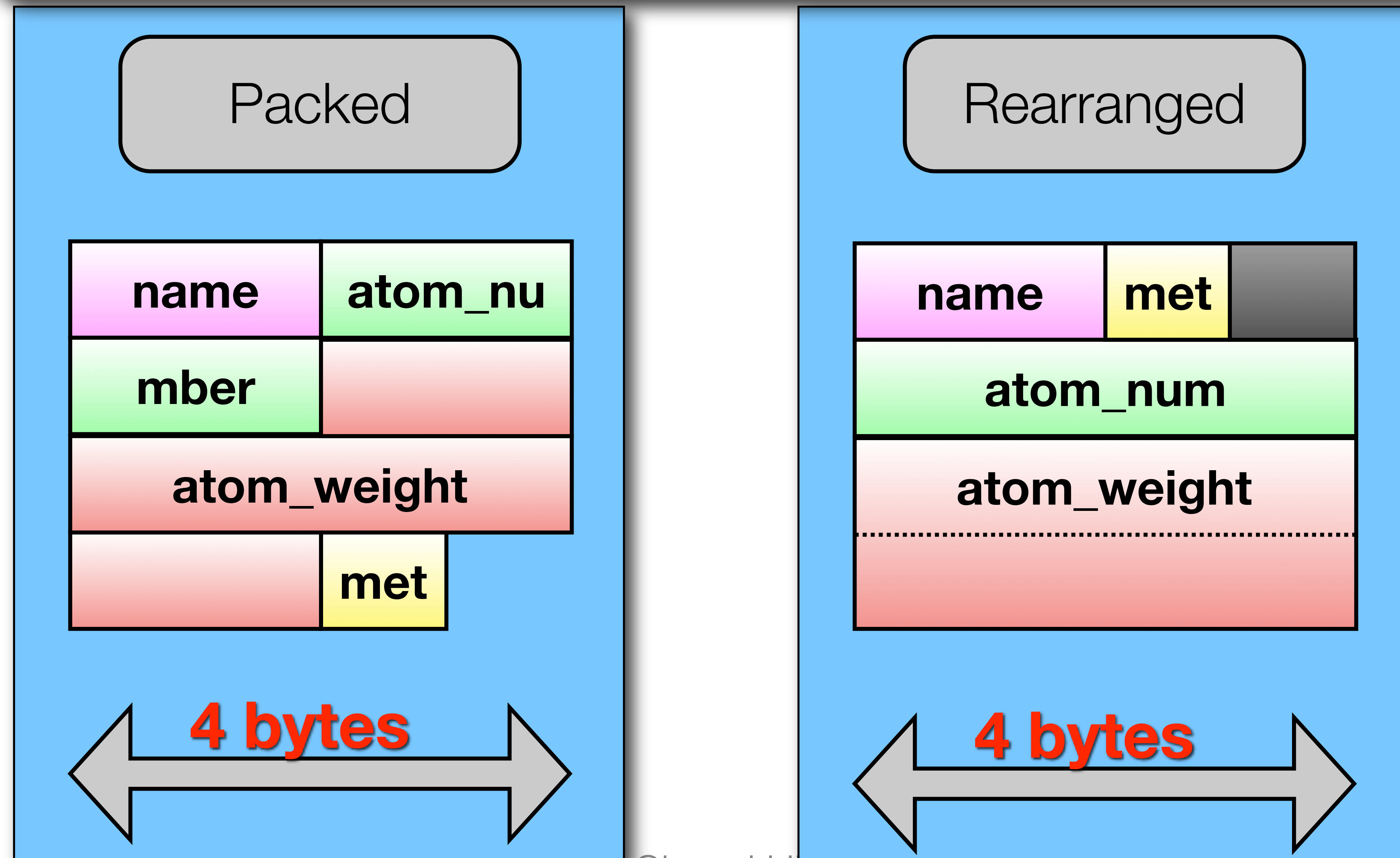



```
type ore = record
    name : two_char;
    atom_num: integer;
    atom_weight: real;
    met: Boolean;
end;
```

Other arrangements



Packed layouts require multiple instructions for **accessing elements** and **assignments**.



Variant Records

- A **variant record** (**union**) provides **two or more alternative fields** or collections of field **but only one bit is valid at any given time**

```
struct element{
  char* Full_name;
  union{
    int atom_num;
    char atom_sym[2];
  }
}
```

element can contain **atom_num** or **atom_sym**, but not both.

Variant Records

```
struct element{
  char* Full_name;
  union{
    int atom_num;
    char atom_sym[2];
  }
}
```