

G22.2110 Programming Languages

6/28/2007
Types and Modules (Ada)

Announcements

Midterm is graded

You can pick it up after class today.

See also:

<http://www.cs.nyu.edu/courses/summer07/G22.2110-001/#grading>
<http://www.cs.nyu.edu/courses/summer07/G22.2110-001/midterm-example-solutions.pdf>

What is a Type?

Point of view	Definition	Example
Denotational	Set of values	(0, 'a'), (1, 'a'), (2, 'a'), ..., (0, 'b'), ..., (-349, 'q'), ...
Constructive	Built-in or composite	<pre>struct s { int _i; char _c; };</pre>
Abstraction	Set of operations	._i, ._c, =, ==, ...

Motivation

Types provide context for operations

E.g., int + int is addition, string + string is concatenation

Operator = built-in function with special syntax

Operand = parameter to operator

Overloading = two different functions have same name, resolved by parameter types

Types restrict what program can do

E.g., string / string is not allowed

Object = something that has lvalue and rvalue

Type clash = applying function to object that does not support it

Type System

Built-in types

Constructors for user-defined types

Mechanisms for associating types with entities

Type checking rules

- Type equivalence rules
- Type compatibility rules
- Type inference rules

Constructors

Constructor defines new type

Competing terminology: constructor creates new object

Not in this lecture (and not in Scott Chapter 7)

Examples

```
typedef char* string;  
int*  
int[4]  
enum year { freshman, sophomore, junior, senior }  
struct Student { string _name; enum year _year; }  
union NumberOrPointer { int _num; void* _ptr; }
```

Associating Types with Entities

Entity: variable, function, type

Declaration

```
introduces name      extern int i;
indicates scope      void f(int n);
                     struct s;
```

gives type (for variable/function) or kind (for type)

Definition

```
is declaration      int v = 42;
fully describes entity void f(int n) { printf("%d\n", n); }
                    struct s { int _i; char _c; };
```

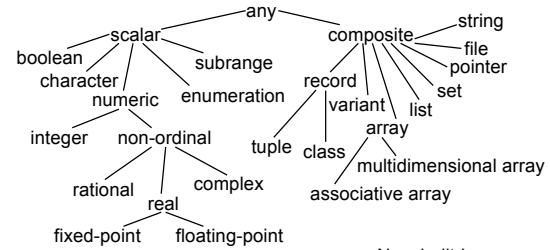
In C: declare before use, define exactly once

Martin Hirzel

G22.2110 NYU 6/28/2007

7

Classification of Types



Also: built-in vs.
user-defined

Martin Hirzel

G22.2110 NYU 6/28/2007

8

Type Equivalence

Equivalence is symmetric compatibility

If types T1 and T2 are equivalent, then program can

use a value of T1 wherever T2 is expected AND
use a value of T2 wherever T1 is expected

E.g., **T1 v1; T2 v2; v1 = v2; v2 = v1;**

Structural equivalence vs. name equivalence

Martin Hirzel

G22.2110 NYU 6/28/2007

9

Structural Equivalence

T1 and T2 are equivalent iff they are

the same primitive type, OR

composed from equivalent types using same constructor

```
struct A1 { int x; double y; };
struct A2 { int x; double y; };
struct A3 { double y; int x; };
struct B1 { int x; struct B1* y; };
struct B2 { int x; struct B2* y; };
```

Martin Hirzel

G22.2110 NYU 6/28/2007

10

Name Equivalence

T1 and T2 are equivalent iff they come from the same definition

Strict vs. loose name equivalence:
is alias equivalent?

```
typedef char* string;
string x;
char* y;
x = y;
y = x;
```

Martin Hirzel

G22.2110 NYU 6/28/2007

11

Type Compatibility

Compatibility is asymmetric substitutability

If type T1 is compatible with T2, then program can use value of T1 wherever T2 is expected

Clean compatibility rules:

object-oriented programming (sub <: super)

subranges (a..b <: c..d iff a≥c and b≤d)

Otherwise, more ad-hoc rules

case-by-case type conversions

Martin Hirzel

G22.2110 NYU 6/28/2007

12

Type Conversions

Example	From	To
<code>3 + 1.5</code>	int	double
<code>int i = 3.145</code>	double	int
<code>3 + (int)1.5</code>	double	int
<code>if (x = malloc(...))</code>	void*	boolean
<code>(int*)malloc(...)</code>	void*	int*
<code>(number->string y)</code>	number	string

lose precision
 implicit (others are explicit)
 change bits (others reinterpret bits)
 dynamic check (others check statically or never)

Coercion = implicit type conversion
 Non-converting cast = reinterprets bit pattern

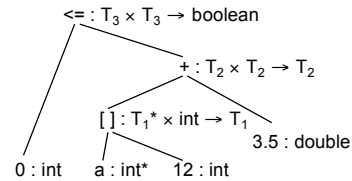
Martin Hirzel

G22.2110 NYU 6/28/2007

13

Type Inference

`0 <= a[12] + 3.5`



Martin Hirzel

G22.2110 NYU 6/28/2007

14

Strong vs. Weak Typing Static vs. Dynamic Typing

Strong: avoids all type clashes

Weak e.g., `int i = 1234; ((int*)i)[0] = 5678;`

Pro strong: prevents silent bugs

Pro weak: enables system programming

Static: checks at compile time

Dynamic e.g., Scheme function `number->string`

Pro static: prevents runtime exception

Pro dynamic: reduces notational and conceptual burden

Martin Hirzel

G22.2110 NYU 6/28/2007

15

Records

Object of record type consists of
 fixed set of heterogeneous subobjects
 (fields)

Denotational: cartesian product

Constructive: in C, use `struct` constructor

Abstraction: `.f, ==, =`

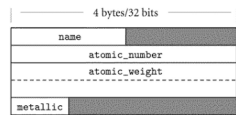
Martin Hirzel

G22.2110 NYU 6/28/2007

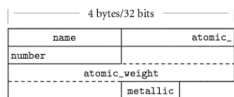
16

Records: Example

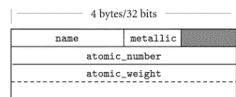
```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
};
```



Scott Fig. 7.1: default layout



Scott Fig. 7.2: packed layout



Scott Fig. 7.3: rearranged

Martin Hirzel

G22.2110 NYU 6/28/2007

17

Variants

Objects of variant type consist of one of
 fixed set of heterogeneous alternatives
 (fields)

Denotational: set union of alternatives

Constructive: in C, use `union` constructor

Abstraction: `.f, ==, =`

Martin Hirzel

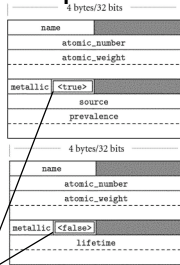
G22.2110 NYU 6/28/2007

18

Variants: Example

```

struct element {
  char name[2];
  int atomic_number;
  double atomic_weight;
  _Bool metallic;
  _Bool naturally_occurring;
  union {
    struct {
      char* source;
      double prevalence;
    } natural_info;
    double lifetime;
  } extra_fields;
};
    
```



tag, aka. discriminant Scott Fig. 7.4: layout

Martin Hirzel

G22.2110 NYU 6/28/2007

19

Arrays

Objects of array type consist of numbered homogeneous subobjects (elements)

Denotational: mapping from index type to element type

Constructive: in C, use [] constructor

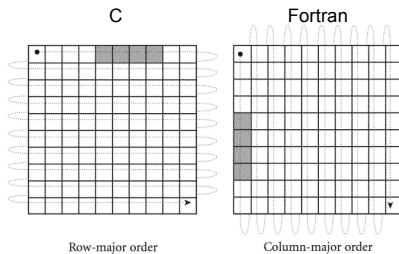
Abstraction: [], ==, =

Martin Hirzel

G22.2110 NYU 6/28/2007

20

Contiguous array layouts



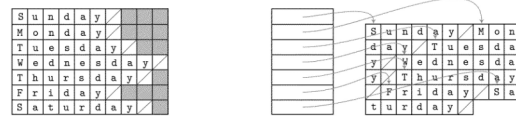
Scott Fig. 7.9: $A[\text{row}][\text{col}]$ with $n\text{Rows}=n\text{Cols}=10$

Martin Hirzel

G22.2110 NYU 6/28/2007

21

Row-pointer array layout



Scott Fig. 7.10: $\text{char days}[][][10]$ vs. $\text{char *days}[10]$

Martin Hirzel

G22.2110 NYU 6/28/2007

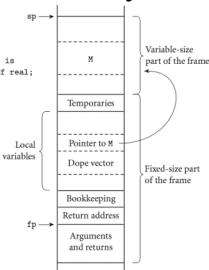
22

Dope Vectors for Arrays

```

procedure foo (size : integer) is
  M : array (1..size, 1..size) of real;
  ...
begin
  ...
end foo;
    
```

- Scott Fig. 7.9
- Conformant array: shape determined at call time from arguments
- Dope vector: fixed-size description of shape



Martin Hirzel

G22.2110 NYU 6/28/2007

23

More on types

Java (7/5/2007):
object orientation
generics

SML (7/19/2007):
rich static type system
type inference
generics

Martin Hirzel

G22.2110 NYU 6/28/2007

24

Modules

Module is self-contained system component with well-defined interface

Module is top-level scope for related types and subroutines

Motivation: “programming in the large”

- Information hiding
- Prevent name space pollution

Martin Hirzel

G22.2110 NYU 6/28/2007

25

int_stacks.ads

```
package Int_Stacks is
  type Int_Stack is private;
  function Create(Capacity : in Integer) return Int_Stack;
  procedure Push(S : in Int_Stack; N : Integer);
  function Pop(S : in Int_Stack) return Integer;
  function Size(S : in Int_Stack) return Integer;
  procedure Destroy(S : in out Int_Stack);
private
  type Int_Array_Value is array (Natural range <>) of Integer;
  type Int_Array is access Int_Array_Value;
  type Int_Stack_Value is record
    Size : Integer;
    Data : Int_Array;
  end record;
  type Int_Stack is access Int_Stack_Value;
end Int_Stacks;
```

Private type: opaque, clients can only use :=, =, or pass as parameter

Private part of spec: used to compute sizes

Martin Hirzel

G22.2110 NYU 6/28/2007

26

int_stacks.adb

```
with Ada.Unchecked_Deallocation;
package body Int_Stacks is
  procedure Free_Int_Stack is new
    Ada.Unchecked_Deallocation(Int_Stack_Value, Int_Stack);
  ...
  procedure Push(S : in Int_Stack; N : Integer) is
    New_Stack : Int_Array;
  begin
    S.Size := S.Size + 1;
    if S.Data'Last <= S.Size then
      New_Stack := new Int_Array_Value(1 .. 2 * S.Data'Last);
      for I in S.Data'Range loop New_Stack(I) := S.Data(I); end loop;
      Free_Int_Array(S.Data);
      S.Data := New_Stack;
    end if;
    S.Data(S.Size) := N;
  end Push;
  ...
end Int_Stacks;
```

Instantiates generics

Martin Hirzel

G22.2110 NYU 6/28/2007

27

Make dependency explicit

Open namespace for unqualified access

driver.adb

```
with Ada.Text_IO; use Ada.Text_IO;
with Int_Stacks; use Int_Stacks;
procedure Driver is
  Buf_Size : constant := 100;
  Line : String(1 .. Buf_Size);
  N, Characters : Integer;
  S : Int_Stack;
  function Starts_With(Whole, Part : String) return Boolean is
  begin
    return Whole'Last >= Part'Last and Whole(1 .. Part'Last) = Part;
  end Starts_With;
begin
  S := Int_Stacks.Create(2);
  ...
  Push(S, N);
  N := Pop(S);
  ...
  Destroy(S);
end Driver;
```

Martin Hirzel

G22.2110 NYU 6/28/2007

28