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?

<table>
<thead>
<tr>
<th>Point of view</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
</table>
| Denotational  | Set of values | (0, ‘a’), (1, ‘a’), (2, ‘a’), ..., (0, ‘b’), ...
|               |            | (-349, ‘q’), ... |
| Constructive  | Built-in or composite | struct s {
|               |               |   int _i;
|               |               |   char _c;
|               |               | }, ...
| 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;
typedef int* int;
enum year { freshman, sophomore, junior, senior };
struct Student { string _name; enum year _year; };
union NumberOrPointer { int _num; void* _ptr; };

Martin Hirzel
Associating Types with Entities

Entity: variable, function, type

Declaration
- introduces name
- indicates scope
- gives type (for variable/function) or kind (for type)

Definition
- is declaration
- fully describes entity

In C: declare before use, define exactly once

Classification of Types

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

Name Equivalence

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

Strict vs. loose name equivalence:
- is alias equivalent?

Typed:
- char* string;
- string x;
- char* y;
- x = y;
- y = x;

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
Type Conversions

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

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

Type Inference

0 <= a[12] + 3.5
<= : T3 x T3 -> boolean
+ : T2 x T2 -> T2
[...]
12 : int
1 : int
3.5 : double

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

Records: Example

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

Records

Object of record type consists of
fixed set of heterogeneous subobjects (fields)
Denotational: cartesian product
Constructive: in C, use struct constructor
Abstraction: .f, ==, =

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, ==, =
Variants: Example

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

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: `[]`, `==`, `=`

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: `[]`, `==`, `=`

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: `[]`, `==`, `=`

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: `[]`, `==`, `=`

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: `[]`, `==`, `=`

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: `[]`, `==`, `=`

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: `[]`, `==`, `=`

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: `[]`, `==`, `=`

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: `[]`, `==`, `=`

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: `[]`, `==`, `=`
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

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;

int_stacks.adb

with Ada.UncheckedReallocation;
package body Int_Stacks is
  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);
      else
        New_Stack := new Int_Array_Value(S.Data);
      end if;
      S.Data := New_Stack;
      S.Data(S.Size) := N;
    end if;
  end Push;
end Int_Stacks;

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 :
  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;