

Types

- › Louden Chapters 6,9
- › Data Types and Abstract Data Types

1

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Arrays as functions

- › $f: U \rightarrow V$ (if U is ordinal type) $f(i)$ array
- › C arrays types can be without sizes array variables must have fixed size

```
int array_max(int a[], int size) // parameters are essentially pointers
{
    int temp, i;
    assert(size > 0);
    temp = a[0];
    for (i=1; i < size; i++)
        { if (a[i] > temp) temp = a[i]; }
    return temp;
}
```

2

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Java arrays

Java arrays are heap allocated and the size can be specified completely dynamically (but once specified can not change unless reallocated)

```
class ArrayTest
{ static int array_max(int[] a)
  { int temp;
    temp = a[0];
    for (int i = 1; i < a.length; i++)
      { if (a[i] > temp) temp = a[i]; }
    return temp;
  }
}
```

```
u = Integer.parseInt(in.readLine());
int[] x = new int[u];
```

3

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Multidimensional Arrays (row major, column major form)

```
int x[10][20]; /* c code */
int array_max(int a[][20], int size);
```

```
int [] [] x = new int [10][20]; /* Java code */
```

4

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Pointers and Arrays in C

```
int a[] = {1,2,3,4,5};
int* p = a;                a[2] is shorthand for *(a+2);
printf("%d\n", *p);
printf("%d\n", *(p+2));
printf("%d\n", *(a+2));
printf("%d\n", 2[a]);

/* all of the above print 3 :-) */
```

5

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Recursive Datatypes in C

in ML:
datatype CharList = EmptyCharList
| CharListNode of char * CharList;

C requires that each data type has a fixed maximum size determined at translation time

Each node
has a fixed size and
they can be strung
together to form
a list of arbitrary size

```
struct CharListNode
{ char data;
  struct CharListNode* next;
};
typedef struct CharListNode* CharList;
```

6

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Using recursive – pointer types

```
CharList cl = (CharList) malloc(sizeof(struct CharListNode));
(*cl).data = 'a';
(*cl).next = 0;

/* this is so common that there is a short-hand notation */
cl->data = 'a';
cl->next = 0;

(*cl).next = (CharList) malloc(sizeof(struct CharListNode));
>(*cl).next.data = 'b'; /* cl->next->data='b'; */
>(*cl).next.next = 0; /* cl->next->next=0; */
```

7

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Type equivalence

- > Structural equivalence
 - > two datatypes are the same if they have the same structure (trickier than it sounds)
- > Name equivalence
 - > two types are the same only if they have the same name

8

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Type equivalence in C, ML

- › C
 - › name equivalence for structs
 - › structure equivalence for everything else
- › ML
 - › type `Age int;` (structural equivalence)
 - › datatype `NewAge = NewAge int;` (name equivalence)

9

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Type checking

- › C, C++ : weak type checking
- › Scheme : strong dynamic type checking
- › Ada : strong type checking – no type inference (very verbose)
- › ML : strong type checking + type inference
 - › flexibility + protection from errors

10

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

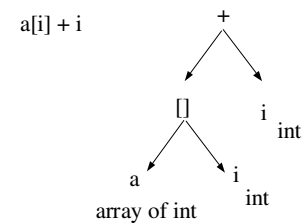
Polymorphic Type Checking

- › Hindley-Milner type checking (1969,1978)
- › Major feature of ML and Haskell
- › The machinery for automatically determining the types of arguments and results of functions without the programmer having to specify them

11

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

An example



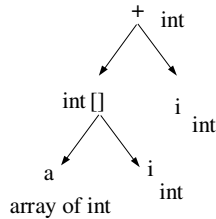
Type checker fills in the types of each node and check if everything is ok

12

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

An example

$a[i] + i$



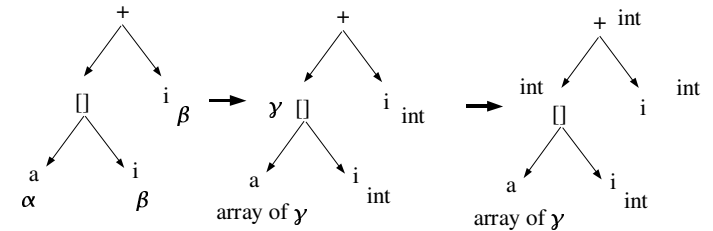
Type checker fills in the types of each node and check if everything is ok

13

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Can we do without type declarations?

type variables: α, β, γ (only int subscripts)



14

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Hindley-Milner type checking

- > Instantiation: once a type variable is replaced by an actual type, the ALL instances of that variable must be updated
- > Unification:
 - > any type variable unifies with any type expression
 - > type constants unify if they are the same
 - > two type constructions unify if they are the same all components also (recursively) unify

15

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Unification examples

- > 'a unifies to b' array
- > int unifies to int
- > 'a list unifies with 'b list
- > 2 can not unify with array of a
- > array of int cannot unify with array of char

16

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Polymorphic type checking

- › 'b array must be the same as 'a array therefore 'b and 'a are the “same” but we don't know what they are
- › REALLY COOL
- › Implicit Parametric Polymorphism (compiler does it for you)
- › Explicit Parametric Polymorphism
- › Ad-hoc polymorphism (overloading)

17

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

There is more to it

- › Read pages 240-244 of your book and using your experience with the ML compiler you should have no problem understanding them

18

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Code translation for polymorphic types

- › Without knowing the types translator cannot determine the size of values
- › Two solutions
 - › Expansion = examine all uses generate separate code for each use
 - › Boxing and tagging = Fix size for scalar types + size field for the rest (flag bit to disambiguate) (also used for garbage collection)

19

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria

Explicit Polymorphism

In ML (explicit works nicely with HM type checking):

```
datatype 'a Stack = EmptyStack
                | Stack of 'a * ('a Stack);
```

In C++ (templates are explicit polymorphism)

```
Stack<int> s;
```

The most recent addition to Java are are Generics which are similar.

```
template <typename T>
struct StackNode
{ T data;
  StackNode<T> *next;
};
template <typename T>
struct Stack
{ StackNode<T>* theStack;
};
```

20

CS330 Spring 2003
Copyright George Tzanetakis, University of Victoria