Types and Type Checking

– lecture 7 –

Types

- Means of providing interpretation to the "bits".

- Two classes:
  - *Basic:* supported in hardware
  - *Aggregate:* hierarchical way of combining basic types

- In *strongly typed languages* (Ocaml, Haskell): means of detecting incorrect usage of functions

Basic Types

- Most languages have these types. We do a case study on C.
- `int` : 32 bits, signed (2's complement)
- `unsigned int` : 32 bits, unsigned
- `long long int` : 64 bits, signed
- `long long unsigned int` : 64 bits, unsigned
- `short int` : 16 bits, signed
- `unsigned short int` : 16 bits, unsigned
- `char` : 8 bits, signed
- `unsigned char` : 8 bits, unsigned
- `unsigned char` : 8 bits, unsigned
- `float` : 32 bit floating point reals
- `double` : 64 bit floating point reals
- `long double` : 80 bit floating point reals
- Operations on these types will be translated directly into the corresponding machine code instructions: *execution is very efficient*

Datatype Conversions

- Usually called *casts*. Two main types:
  - A notion of *value* can be preserved
    - integral $\rightarrow$ real : preserve the value
    - real $\rightarrow$ integral : truncates away the fractional part
    - small size integral $\rightarrow$ large size integral : *sign extended*
      - Conversion may take the `floor` or the `ceiling`, depending on the language.
  - Bits can be copied over.
    - Conversion between pointer types
    - If sizes are not the same, the least significant bytes are usually preserved
- Implicit casts:
  - Compiler tries to perform conversions to match operators and declarations of functions.
  - When the result of an expression does not fit into 32 bits, it is cast to its address.
  - Exception: structures
  - Specific to C, most other languages do not perform this cast

Datatype Conversions

- Usually called *casts*. Two main types:
  - A notion of *value* can be preserved
    - integral $\rightarrow$ real : preserve the value
    - real $\rightarrow$ integral : truncates away the fractional part
    - small size integral $\rightarrow$ large size integral : *sign extended*
      - Conversion may take the `floor` or the `ceiling`, depending on the language.

```
int a; float b;
...
x = a + b ;
```

`a` is converted to float, value is preserved

  - Bits can be copied over.
    - Conversion between pointer types
    - If sizes are not the same, the least significant bytes are usually preserved

- Implicit casts:
  - Compiler tries to perform conversions to match operators and declarations of functions.
  - When the result of an expression does not fit into 32 bits, it is cast to its address.
  - Exception: structures
  - Specific to C, most other languages do not perform this cast

Datatype Conversions

- Usually called *casts*. Two main types:
  - A notion of *value* can be preserved
    - integral $\rightarrow$ real : preserve the value
    - real $\rightarrow$ integral : truncates away the fractional part
    - small size integral $\rightarrow$ large size integral : *sign extended*
      - Conversion may take the `floor` or the `ceiling`, depending on the language.

```
char a; int b;
...
x = a + b ;
```

`a` is sign-extended to int, value is preserved

  - Bits can be copied over.
    - Conversion between pointer types
    - If sizes are not the same, the least significant bytes are usually preserved

- Implicit casts:
  - Compiler tries to perform conversions to match operators and declarations of functions.
  - When the result of an expression does not fit into 32 bits, it is cast to its address.
  - Exception: structures
  - Specific to C, most other languages do not perform this cast

Datatype Conversions

- Usually called *casts*. Two main types:
  - A notion of *value* can be preserved
    - integral → real : preserve the value
    - real → integral : truncates away the fractional part
    - small size integral → large size integral : *sign extended*
      - Conversion may take the `floor` or the `ceiling`, depending on the language.
  - Bits can be copied over.
    - Conversion between pointer types
    - If sizes are not the same, the least significant bytes are usually preserved

```
float a;
...
int f(int x) ;
...
x = f(a);
```

`a` is truncated to int, value is preserved to utmost extent possible

- Implicit casts:
  - Compiler tries to perform conversions to match operators and declarations of functions.
  - When the result of an expression does not fit into 32 bits, it is cast to its address.
  - Exception: structures
  - Specific to C, most other languages do not perform this cast

Aggregate Types

- Arrays

- Most langauges provide *records*
  - In C they are called *structures*
  - In object oriented programming they are extended to *objects*

- Unions: specific to C, help save space.

- High-level aggregate datatypes (Python, Ruby):
  - Lists
  - Tuples
  - Sets
  - Dictionaries
  - implemented in libraries for languages without these primitives

Aggregate Types

[C]

- Arrays
- Most langauges provide *records*
  - In C they are called *structures*
  - In object oriented programming t
- Unions: specific to C, help save spac
- High-level aggregate datatypes (Pyth
  - Lists
  - Tuples
  - Sets
  - Dictionaries
  - implemented in libraries for languages without these primitives

```
struct struct_name {
  type1 field1 ;
  type2 field2, field3 ;
  type3 field4[10] ;
} var1, var2, * var3 ;
```

Aggregate Types

- Arrays

- Most langauges provide *records*
  - In C they are called *structures*
  - In object oriented programming they are extended to *objects*

- Unions: specific to C, help save space.

```
union union_name {
  type1 field1 ;
  type2 field2, field3 ;
  type3 field4[10] ;
} var1, var2, * var3 ;
```

- High-level aggregate datatypes (Python
  - Lists
  - Tuples
  - Sets
  - Dictionaries
  - implemented in libraries for languages without these primitives

C Pointers

- Models the memory address of a datum.
- Memory is an array of bytes (unsigned characters) $\rightarrow$ pointer is an unsigned integer.
- At the type system level, the language distinguishes between pointers and integers for safety reasons.
  - This does not happen in VAL
- Declaration: `type * p`
- Operations:
  - `*p` : dereference
  - `p+k` : pointer arithmetic, points `k*sizeof(*p)` bytes away.
- The address of any *lvalue* can be captured into a pointer with the `&` (address-of) operator.

Pointers to Functions


```
type f() {
  ...
}

...
{
  type (*p)() ; // pointer to a function that returns "type"
  p = &f ; // assign p to address of f
  ...
  (*p)() ;  // calls f
  ...
}
```

Pointers to Functions


```
type f() {
  ...
}

...
{
  type (*p)() ; // pointer to a function that returns "type"
  p = &f ; // assign p to address of f
  ...
  (*p)() ;  // calls f
  ...
}
```

p=f would work too

Pointers to Functions


```
type f() {
  ...
}

...
{
  type (*p)() ; // pointer to a function that returns "type"
  p = &f ; // assign p to address of f
  ...
  (*p)() ;  // calls f
  ...
}
```

`p=f` would work too

`f` would be implicitly cast to its address

Pointers to Functions


```
type f() {
  ...
}

...
{
  type (*p)() ; // pointer to a function that returns "type"
  p = &f ; // assign p to address of f
  ...
  (*p)() ;  // calls f
  ...
}
```

`p=f` would work too

`f` would be implicitly cast to its address

`p()` would work too

Pointers to Functions


```
type f() {
  ...
}

...
{
  type (*p)() ; // pointer to a function that returns "type"
  p = &f ; // assign p to address of f
  ...
  (*p)() ;  // calls f
  ...
}
```

`p=f` would work too

`f` would be implicitly cast to its address

`p()` would work too

`p` would be implicitly cast to its dereference

A Rich Type Language


```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

A Rich Type Language


```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is

A Rich Type Language

[C]

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function

A Rich Type Language


```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function that returns a pointer

A Rich Type Language

[C]

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function that returns a pointer to a function

A Rich Type Language


```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function that returns a pointer to a function that returns a pointer

A Rich Type Language


```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

f is a function that returns a pointer to a function that returns a pointer to int

A Rich Type Language

[C]

Type operators have precedence: () and [] bind tighter than * ; we can use brackets to alter the precedence.

int *a[10] ; declares an array of pointers to int
int (*a)[10] ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

A Rich Type Language

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

A Rich Type Language

Type operators have precedence: () and [] bind tighter than * ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

A Rich Type Language

[C]

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

`a` is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

A Rich Type Language

[C]

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

`a` is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

A Rich Type Language

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

A Rich Type Language

[C]

```
#include <stdlib.h>

int * g(int a, int b) { // function tha
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;
```

Type operators have precedence: `()` and `[]` bind tighter than `*` ; we can use brackets to alter the precedence.

`int *a[10]` ; declares an array of pointers to int
`int (*a)[10]` ; declares a pointer to an array of ints

a is a pointer to a pointer to an array of pointers to arrays of ints

```
int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

A Rich Type Language

[C]

```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Each layer of pointers must be intialized.

A Rich Type Language


```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Drop the a to obtain the cast.

A Rich Type Language


```
#include <stdlib.h>

int * g(int a, int b) { // function that returns pointer to int
  return (int*)malloc(10) ;
}

typedef int * t(int,int) ;

int * (*f())(int,int) { // function that returns the address of g
  return & g ;
}

int main() {
  int (*(**a)[3])[10] ;
  a = (int (*(**)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  *a = (int (*(*)[])[10])malloc(sizeof(int (*[3])[10])) ;
  *(a+1) = (int (*(*)[])[10])malloc(sizeof(int (*(*)[])[10])) ;
  (**a)[0] = (int (*)[10])malloc(sizeof(int [10])) ;
  (*(**a)[0])[1] = 100 ;
  (*f())(1,2) ; // calls g(1,2)
  return 0 ;
}
```

Move one * from right side to the left side

Continue on for each level of pointers

Type Checking for a Toy Language

Type constructors:

`int x ;`	: declares `x` to be of base type integer
`T $ x ;`	: `x` is a pointer to type `T`
`T x @ N ;`	: `x` is an array of size `N`, with elements of type `T`
`T x#(T1 arg1; ...;Tn argn) ;`	: `x` is a function that takes n arguments, of types `T1,...,Tn`. Arguments must be separated by semicolon
`Str ~~~ { ... } ;`	: declares the structure `Str`, having the fields declared as normal 'variables' within the curly braces.
`Str ~~~ x, y, z ;`	: declare `x, y, z` to be of the structure type `Str`. `Str` must have been declared in advance.

Type-related Operators

`$` : pointer dereference -- `($p)` same as `(*p)` in C

`&` : address-of operator, similar to C

`_@N` : array indexing -- `a@2` same as `a[2]` in C

`_#_` : function application -- `f#(2;3)` same as `f(2,3)` in C

`_~~_` : field selector from a structure -- `s~~f` same as `s.f` in C

Code Sample

```
int x,             % x is an integer
    $y,            % y is a pointer to an integer
    z@10,          % z is an array of 10 integers
    $ $ $ zzz ;    % zzz is a pointer to pointer to pointer to integer
```

```
int ($fptr)#( int aa;               % fptr is a pointer to a function that
              int bb;               % takes 3 arguments and returns an integer
              int ($pp)#(int x)) ;  % -- the third arg is a pointer to function too
```

```
s ~~~ { int f1,$ f2@10 ;      % s is a structure type, with 4 fields: an integer,
        int e1,e3@10 } ;      %  an array of pointers to intgers, a second integer,
                              %  and an array of integers.
```

```
s ~~~ p,q ;                   % p and q are variables of type s
```

Code Sample

```
int proc # ( int a;            % proc is a procedure that takes 3 arguments
             int b;            % and returns an int. The third arg has a function type
             int ($p)#(int x)) :: {
   int c,d;                    % two local integer variables
   int f # (int w) ::          % a local (nested) function, can only be called from inside proc
        {x = w ; return 1 } ;

   if a > 100 then {
        return a+1             % the type of the return expression will be checked against
   } ;                         % the type declared to be returned by the procedure

   ss ~~~ { int aa,bb } ;      % declarations can appear anywhere; another structure with 2 fields
   ss ~~~ sss ;                % sss is a structure of type ss

   sss ~~ aa =                 % assignment to the field 'aa' of structure 'sss'
      2 + c +
       f#d * proc#(x;$y;p) ;   % calling f and then proc, recursively

   return ($pp)#(3) ;          % calling the pointer to function, and returning its value from proc
} ;
```

Code Sample

```
int qq#(int d) ::                % another function
    { return d+1 } ;

fptr = & proc ;                  % assignment of the address of proc to a pointer to function

z@8 = ($fptr)#(6;20;&qq) ;       % calling a pointer to function

x = 1 ; y = & x ; z@1 = $y ;     % examples of assignments, some involving function calls
x = proc#(3;4;&qq) ;             %  -- all data transfers are int
$ y = proc#(z@2;$y;&qq) ;
p ~~ f1 = q ~~ e1 ;              % assignment of and to structure fields

x = $ (z+2) ;                    % pointer arithmetic, z+2 is a legal pointer

$ $ zzz = y ;                    % multiple dereferencing, a pointer to int is assigned

y = (p~~f2)@2 ;                  % access to an array element inside a structure

$ $ $ zzz = $((p~~f2)@2)         % multiple dereferencing of pointers
```

Type Checking of Toy Language

```
/*
 * The main predicate computes the type of an
 * expression/statement. For expressions (i.e. syntactic constructs
 * that have value, the type is the type of the value. For type
 * declarations, the type returned is the atom 'type', and for all
 * other statements, the type is 'stmt'. The dummy types 'type' and
 * 'stmt' are important so as to allow us to check that structures
 * do not have any code in their definition.
 *
 * The arguments are as follows:
 *   arg1 (input)  : expression whose type is being calculated
 *   arg2 (output) : the type returned for arg1
 *   arg3 (input)  : the input attributes
 *   arg4 (output) : the output attributes
 *
```

Type Representation

```
* Types are encoded as an "inside-out" representation of
* the declaration. For instance, a declaration of the form
*
*   int ($fptr)#( int aa; int bb; int ($pp)#(int x)) ;
*
* associates the following type representation with variable fptr :
*
*   $(int#(int;int;$(int#int)))
*
* This representation has the syntactic tree
*
*          $
*          |
*          #
*         / \
*      int   ;
*           / \
*       int    ;
*             / \
*          int   $
*                |
*                #
*               / \
*            int  int
```

Variables, Constants, and Pointers

```
typecheck(X,T,A,A) :- atom(X), !, get_assoc(type_env,A,Local), member((X->T),Local),!.
          % if X is a variable, then it must have been declared, so its name
          % and associated type must be stored in the type_env attribute

typecheck(N,int,A,A) :- integer(N), !.
          % an integer constant has type 'int'

typecheck(E,($T),A,A) :- E =.. [Op,E1,E2], member(Op,[+,-]),
    typecheck(E1,TE,A,_),  % pointer arithmetic: if p is a pointer, then p+k and p-k are legal pointers
    same_type(TE,$T),      % same_type checks if a cast is possible, so that E1 is interpreted as ptr
    typecheck(E2,int,A,_). % array names will be cast to pointers.

typecheck(E,int,A,A) :- E =.. [Op,E1,E2], % arithmetic expressions have integer operands and result
    member(Op,[+,-,*,/,rem,/\,\/,==,\=,=<,>=,<,>]),!, % binary operators are checked here
    typecheck(E1,int,A,_), typecheck(E2,int,A,_).

typecheck(E,int,A,_) :- % arithmetic expressions have integer operands and result
    E =.. [Op,E1], member(Op,[\,-,+]), !,  % unary operators are checked here
    typecheck(E1,int,A,_).

typecheck(($E),T,A,A) :- !, % type of a dereferenced expression: the type of E must be
    typecheck(E,TE,A,_),    %   equivalent to that of pointer to some type T
    same_type($T,TE).       %   and then the type of the result will be T.
```

Arrays, Addresses, Proc Calls, Struct Access

```
typecheck((E1@E2),T,A,A) :-!,       % array subscript expression
    typecheck(E1,TE,A,_),           %   the type of E1 must be equivalent to that of array with
    same_type(TE,@(T)),             %   elements of some type T, and then the result will be
    typecheck(E2,int,A,_).          %   of type T. The subscript E2vmust be of type int

typecheck((&E),($T),A,A) :-!,        % address-of expression
    (     atom(E) ; E = _ @ _        % operator & can only be applied to a variable, an array element,
    ;     E = _ ~~ _ ; E = $_ ),     % a structure field, or a dereferenced pointer
    typecheck(E,T,A,_).

typecheck((P#L),T,A,A) :- !,         % function call expression
    typecheck(args(L),LT1,A,_),      % compute the type of all args as a tuple LT1
    typecheck(P,(T#LT2),A,_),        % the type of P recorded in type_env must be of the pattern
    same_type(LT1,LT2).              % T#LT2, where T is some return type, and LT2 is some tuple type
                                     % Then, LT1 and LT2 must be cast-able to each other, and T will be
                                     % the type of the function call expression.

typecheck(args(L),TL,A,A) :- !,      % tuple of arguments. 'args' is just a wrapper to avoid confusion
    (   L = (H;T), TL = (TH;TT)      % with statement sequencing { S1 ; S2 }, because the ';' is used there too
    ->  typecheck(H,TH,A,_),         % if the tuple has more than 1 component, compute the types by
        typecheck(args(T),TT,A,_)% recursing through all components
    ;   typecheck(L,TL,A,_) ).       % otherwise, just compute the type of the singleton argument

typecheck((S ~~ F),T,A,A) :- !,             % structure field access; F must be an atom, and S must be
    atom(F), typecheck(S,~~~(TS),A,_),      % a declared structure. 'check_struct_acc' will check if
    check_struct_acc(TS,F,T,A).             % the field was declared, and retrieve its type.
```

Statements

```
% Type checking for statements. The returned type is either the atom
% 'type', that represents the fact that the program fragment subjected
% to type checking has only declarations (useful to check that the
% inside of a structure does not have any executable code -- which would
% not be syntactically acceptable); or the atom 'stmt', to represent
% that it does contain some executable code.

typecheck((E1 = E2),stmt,A,A) :- !, % assignment
    typecheck(E1,T1,A,_),           % the types of the two sides must be cast-able to each other
    typecheck(E2,T2,A,_),
    (   atom(E1) ;   E1 = _ @ _     % the LHS must be a l-value, i.e. a variable, array element,
    ;   E1 = $ _ ;   E1 = _ ~~ _ ), % dereferenced expression, or structure field
    same_type(T1,T2),
    writeln( 'Assignment' : E1=E2 | T1 = T2 ). % We print all the assignments, to have some verification

% 'if' and 'while' statements are straightforward. The inner scopes may
% have their own declarations, but these do not need to survive as we
% return to the outer scope, thus we do not care about the output attrs

typecheck((if B then { S }),stmt,A,A) :- !,
    typecheck(B,int,A,_), typecheck(S,stmt,A,_).

typecheck((if B then { S1 } else { S2 }), stmt, A, A) :-
    typecheck(B,int,A,_), typecheck(S1,stmt,A,_), typecheck(S2,stmt,A,_).

typecheck((while B then { S }),stmt,A,A) :- !, typecheck(B,int,A,_), typecheck(S,stmt,A,_).
```

Declarations and Misc

```
% Declarations must be stored in the type environment. All
% variable declarations start with 'int' or have ~~~ at the top level,
% so they are easy to identify

typecheck(int Decl, type, Ain, Aout) :- !,
    process_decl(int,Decl,Ain,Aout,0). % places the declared identifier in the type environment

typecheck( Str ~~~ { Fields } , type, Ain, Aout) :- !, % structure declaration
    get_assoc(type_env,Ain,Orig_env,A1,[]),            % start with a fresh type environment, but preserve the old one
    typecheck(Fields,type,A1,A2),                      % the {...} block of the structure must not contain executable code
                                                       %  -- this is enforced by expected the atom 'type' as return type
    get_assoc(type_env,A2,StructFields,A3,Orig_env),   % convert the computed type environment into a list of fields for the
                                                       % structure and restore the original type environment
    get_assoc(structures,A3,Structures,Aout,[(Str->StructFields)|Structures]).
                                                       % The output attributes are important here

typecheck(Str ~~~ Decls, type, Ain, Aout) :- !, % variable declaration, whose type is a previously defined structure
    get_assoc(structures,Ain,Structures),       % check that the structure has been declared
    member(Str->_,Structures),
    process_decl((~~~(Str)),Decls,Ain,Aout,0).  % process the declaration, placing the declared variable in the type
                                                % environment

typecheck({S},T,A,A) :- !, typecheck(S,T,A,_).  % handle local scope

typecheck((S;),T,A,A) :- !, typecheck(S,T,A,_). % handle statements terminated with ';'

typecheck((S1 ; S2), T, Ain, Aout) :- !,   % sequence of statements, attributes must be handled properly
    typecheck(S1,T1,Ain,Aaux),
    typecheck(S2,T2,Aaux,Aout),
    (   T1 = type, T2 = type, T = type, !  % compute the type of the statement; return 'type' only if both S1 and S2
    ;   T = stmt )   .                     % have returned 'type' -- means there's no executable code in S1 and S2
```

Procedure Definition and Return

```
typecheck(Head::{Body},stmt,Ain,Aout) :- !,           % procedure definition
    Head =.. [T,Rest],
    process_decl(T,Rest,Ain,A1,0),                   % compute the type of the function id, and place it in the type env
    get_assoc(type_env,A1,[_->Type#_|_]),            % retrieve the return type of the function, and store it in the
    get_assoc(return_type,A1,OrigRetType,A2,Type),   % 'return_type' attribute; save the original value so it can be
                                                     % returned later

    typecheck({Body},stmt,A2,A3),                    % type-check the body; return expressions will be matched against
                                                     % val of return_type

    put_assoc(return_type,A3,OrigRetType,Aout).      % restore the original return type, in case we're in a nested function

typecheck(return Expr,stmt,A,A) :- !,  % return statement
    typecheck(Expr,Type,A,_),          % type check the return expression, and match it against the return_type attribute
    get_assoc(return_type,A,T),
    same_type(Type,T).                 % The function's declared return type and the type of the return expression
                                       % must be cast-able
```

Conclusion

- Types are trees, and type checking is pattern matching
- Multiple base types: implicit/explicit cast operators must be added to the language
- Cast-ability can be decided via pattern matching as well.